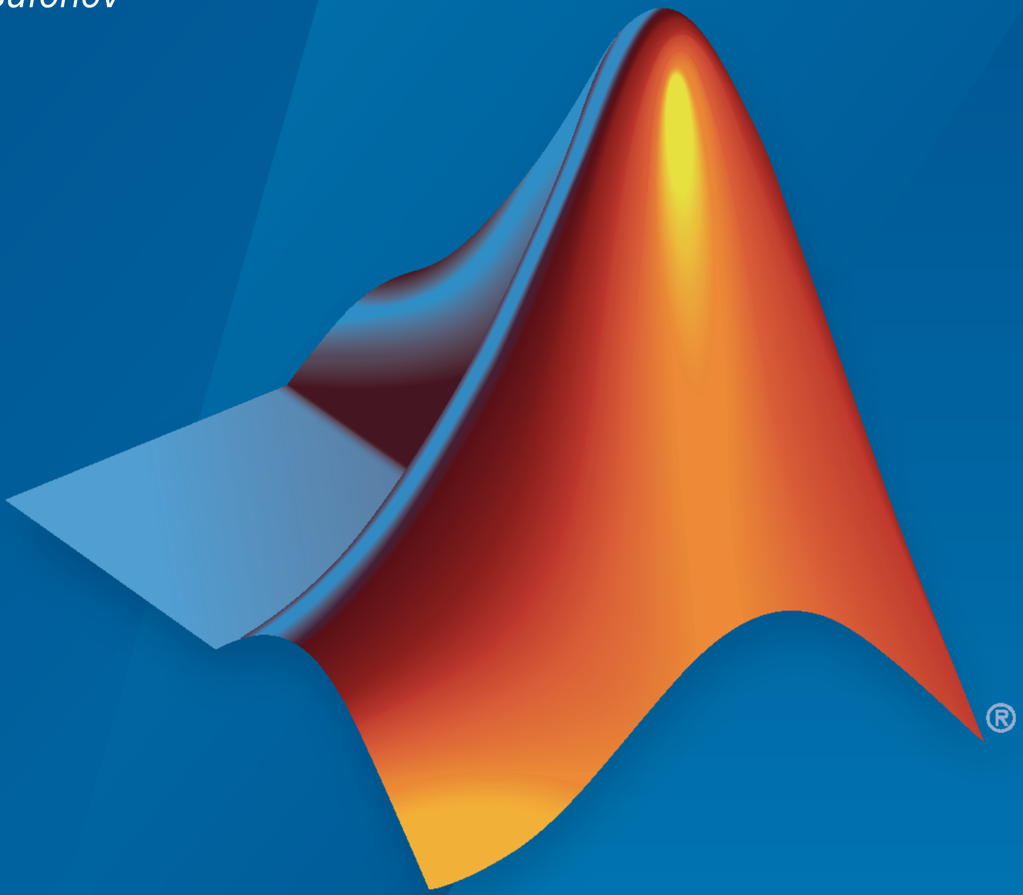
Robust Control Toolbox™

User's Guide

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov



MATLAB®

R2020**a**

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Robust Control Toolbox™ User's Guide

Trademarks

Patents

Revision History

September 2005	First printing	New for Version 3.0.2 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.4.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.5 (Release 2010b)
April 2011	Online only	Revised for Version 3.6 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)
March 2014	Online only	Revised for Version 5.1 (Release 2014a)
October 2014	Online only	Revised for Version 5.2 (Release 2014b)
March 2015	Online only	Revised for Version 5.3 (Release 2015a)
September 2015	Online only	Revised for Version 6.0 (Release 2015b)
March 2016	Online only	Revised for Version 6.1 (Release 2016a)
September 2016	Online only	Revised for Version 6.2 (Release 2016b)
March 2017	Online only	Revised for Version 6.3 (Release 2017a)
September 2017	Online only	Revised for Version 6.4 (Release 2017b)
March 2018	Online only	Revised for Version 6.4.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.5 (Release 2018b)
March 2019	Online only	Revised for Version 6.6 (Release 2019a)
September 2019	Online only	Revised for Version 6.7 (Release 2019b)
March 2020	Online only	Revised for Version 6.8 (Release 2020a)

Contents

1 Building Uncertain Models

2 Generalized Robustness Analysis

3 Mu Synthesis

4 Introduction to Linear Matrix Inequalities

5 LMI Lab

6 Analyzing Uncertainty Effects in Simulink

Building Uncertain Models

Introduction to Uncertain Elements

Uncertain elements (also called uncertain Control Design Blocks (Control System Toolbox)) are the building blocks used to form uncertain matrix objects and uncertain system objects. There are six types of uncertain blocks, summarized in the following table.

Function	Description
ureal	Uncertain real parameter on page 1-4
ultidyn	Uncertain, linear, time-invariant dynamics on page 1-9
umargin	Uncertain gain and phase on page 1-12
ucomplex	Uncertain complex parameter on page 1-15
ucomplexm	Uncertain complex matrix on page 1-15
udyn	Unmodeled dynamics on page 1-19

To build models uncertain systems, you combine these control design blocks with fixed dynamic elements to create uncertain state-space (uss) models.

All of the elements have properties, which are accessed through get and set methods. This get and set interface mimics the Control System Toolbox™ and MATLAB® Handle Graphics® behavior. For instance, get(a,'PropertyName') is the same as a.PropertyName, and set(b,'PropertyName',Value) is the same as b.PropertyName = value. Functionality also includes tab-completion and case-insensitive, partial name property matching.

For ureal, ucomplex and ucomplexm elements, the syntax is

```
p1 = ureal(name,NominalValue,Prop1,val1,Prop2,val2,...);
p2 = ucomplex(name,NominalValue,Prop1,val1,Prop2,val2,...);
p3 = ucomplexm(name,NominalValue,Prop1,val1,Prop2,val2,...);
```

For ultidyn and udyn, the NominalValue is fixed, so the syntax is

```
p4 = ultidyn(name,ioSize,Prop1,val1,Prop2,val2,...);
p5 = udyn(name,ioSize,Prop1,val1,Prop2,val2,...);
```

For umargin blocks, you provide the range of gain variation you want to model. umargin interprets this range as a disk-based gain margin. To get a disk-based gain margin from a target gain and phase variation, use getDGM.

```
DGM = getDGM(GM,PM,'balanced);
p6 = umargin(name,DGM,Prop1,val1,...);
```

For ureal, ultidyn, umargin, ucomplex and ucomplexm elements, the command usample will generate a random instance (i.e., not uncertain) of the element, within its modeled range. For example,

```
usample(p1)
```

creates a random instance of the uncertain real parameter p1. With an integer argument, whole arrays of instances can be created. For instance

```
usample(p4,100)
```

generates an array of 100 instances of the `ultidyn` object `p4`. See “Generate Samples of Uncertain Systems” on page 1-50 to learn more about `usample`.

See Also

`ultidyn` | `umargin` | `ureal`

Related Examples

- “Create Models of Uncertain Systems”
- “Uncertain Real Parameters” on page 1-4
- Uncertain gain and phase on page 1-12
- “Uncertain LTI Dynamics Elements” on page 1-9

Uncertain Real Parameters

An uncertain real parameter, `ureal`, is the Control Design Block (Control System Toolbox) that represents a real number whose value is uncertain.

Properties of Uncertain Real Parameters

Uncertain real parameters have a name (the `Name` property), and a nominal value (the `NominalValue` property). Several other properties (`PlusMinus`, `Range`, `Percentage`) describe the uncertainty in parameter values.

All properties of a `ureal` can be accessed through `get` and `set`. The properties are:

Properties	Meaning	Class
`Name`	Internal name	`char`
`NominalValue`	Nominal value of element	`double`
`Mode`	Signifies which description (from`'PlusMinus'`, `'Range'`, `'Percentage'`) of uncertainty is invariant when `NominalValue` is changed	`char`
`PlusMinus`	Additive variation	`scalar or 1x2` `double`
`Range`	Numerical range	`1x2` `double`
`Percentage`	Additive variation (% of absolute value of nominal)	`scalar or 1x2 double`
`AutoSimplify`	`'off' | {'basic'} |'full'`	`char`

The properties `Range`, `Percentage` and `PlusMinus` are all automatically synchronized. If the nominal value is 0, then the `Mode` cannot be `Percentage`. The `Mode` property controls what aspect of the uncertainty remains unchanged when `NominalValue` is changed. Assigning to any of `Range`/`Percentage`/`PlusMinus` changes the value, *but does not* change the mode.

The `AutoSimplify` property controls how expressions involving the real parameter 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'` (model-reduction-like techniques are applied). See “Simplifying Representation of Uncertain Objects” on page 1-47 to learn more about the `AutoSimplify` property and the command `simplify`.

If no property/value pairs are specified, default values are used. The default `Mode` is `PlusMinus`, and the default value of `PlusMinus` is `[-1 1]`. Some examples are shown below. In many cases, the full property name is not specified, taking advantage of the case-insensitive, partial name property matching.

Create Uncertain Real Parameters

This example shows how to create uncertain real parameters, modify properties such as range of uncertainty, and sample uncertain parameters.

Create an uncertain real parameter, nominal value 3, with default values for all unspecified properties (including plus/minus variability of 1).

```
a = ureal('a',3)
```

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

View the properties and their values, and note that the `Range` and `Percentage` descriptions of variability are automatically maintained.

```
get(a)
```

```
    NominalValue: 3
            Mode: 'PlusMinus'
           Range: [2 4]
       PlusMinus: [-1 1]
      Percentage: [-33.3333 33.3333]
    AutoSimplify: 'basic'
            Name: 'a'
```

Create an uncertain real parameter, nominal value 2, with 20% variability. Again, view the properties, and note that the `Range` and `PlusMinus` descriptions of variability are automatically maintained.

```
b = ureal('b',2,'Percentage',20)
```

```
b =
  Uncertain real parameter "b" with nominal value 2 and variability [-20,20]%.
```

```
get(b)
```

```
    NominalValue: 2
            Mode: 'Percentage'
           Range: [1.6000 2.4000]
       PlusMinus: [-0.4000 0.4000]
      Percentage: [-20 20]
    AutoSimplify: 'basic'
            Name: 'b'
```

Change the range of the parameter. All descriptions of variability are automatically updated, while the nominal value remains fixed. Although the change in variability was accomplished by specifying the `Range`, the `Mode` is unaffected, and remains `Percentage`.

```
b.Range = [1.9 2.3];
get(b)
```

```
    NominalValue: 2
            Mode: 'Percentage'
           Range: [1.9000 2.3000]
       PlusMinus: [-0.1000 0.3000]
      Percentage: [-5.0000 15.0000]
    AutoSimplify: 'basic'
            Name: 'b'
```

As mentioned, the `Mode` property signifies what aspect of the uncertainty remains unchanged when `NominalValue` is modified. Hence, if a real parameter is in `Percentage` mode, then the `Range` and `PlusMinus` properties are determined from the `Percentage` property and `NominalValue`. Changing `NominalValue` preserves the `Percentage` property, and automatically updates the `Range` and `PlusMinus` properties.

```
b.NominalValue = 2.2;
get(b)
```

```
    NominalValue: 2.2000
            Mode: 'Percentage'
           Range: [2.0900 2.5300]
       PlusMinus: [-0.1100 0.3300]
      Percentage: [-5.0000 15.0000]
    AutoSimplify: 'basic'
            Name: 'b'
```

Create an uncertain parameter with an asymmetric variation about its nominal value. Examine the properties to confirm the asymmetric range.

```
c = ureal('c',-5,'Percentage',[-20 30]);
get(c)
```

```
    NominalValue: -5
            Mode: 'Percentage'
           Range: [-6 -3.5000]
       PlusMinus: [-1 1.5000]
      Percentage: [-20 30]
    AutoSimplify: 'basic'
            Name: 'c'
```

Create an uncertain parameter, specifying variability with `Percentage`, but force the `Mode` to be `Range`.

```
d = ureal('d',-1,'Mode','Range','Percentage',[-40 60]);
get(d)
```

```
    NominalValue: -1
            Mode: 'Range'
           Range: [-1.4000 -0.4000]
       PlusMinus: [-0.4000 0.6000]
      Percentage: [-40 60]
    AutoSimplify: 'basic'
            Name: 'd'
```

Finally, create an uncertain real parameter, and set the `AutoSimplify` property to `'full'`.

```
e = ureal('e',10,'PlusMinus',[-23],'Mode','Percentage','AutoSimplify','Full')
```

```
e =
  Uncertain real parameter "e" with nominal value 10 and variability [-230,230]%.
```

```
get(e)
```

```
    NominalValue: 10
            Mode: 'Percentage'
           Range: [-13 33]
       PlusMinus: [-23 23]
      Percentage: [-230 230]
    AutoSimplify: 'full'
            Name: 'e'
```

Specifying conflicting values for `Range/Percentage/PlusMinus` when creating a `ureal` element does not result in an error. In this case, the last specified property is used. This last occurrence also determines the `Mode`, unless `Mode` is explicitly specified, in which case that is used, regardless of the property/value pairs ordering.

```
f = ureal('f',3,'PlusMinus',[-2 1],'Percentage',40)
```

```
f = 
  Uncertain real parameter "f" with nominal value 3 and variability [-40,40]%.
```

```
g = ureal('g',2,'PlusMinus',[-2 1],'Mode','Range','Percentage',40)
```

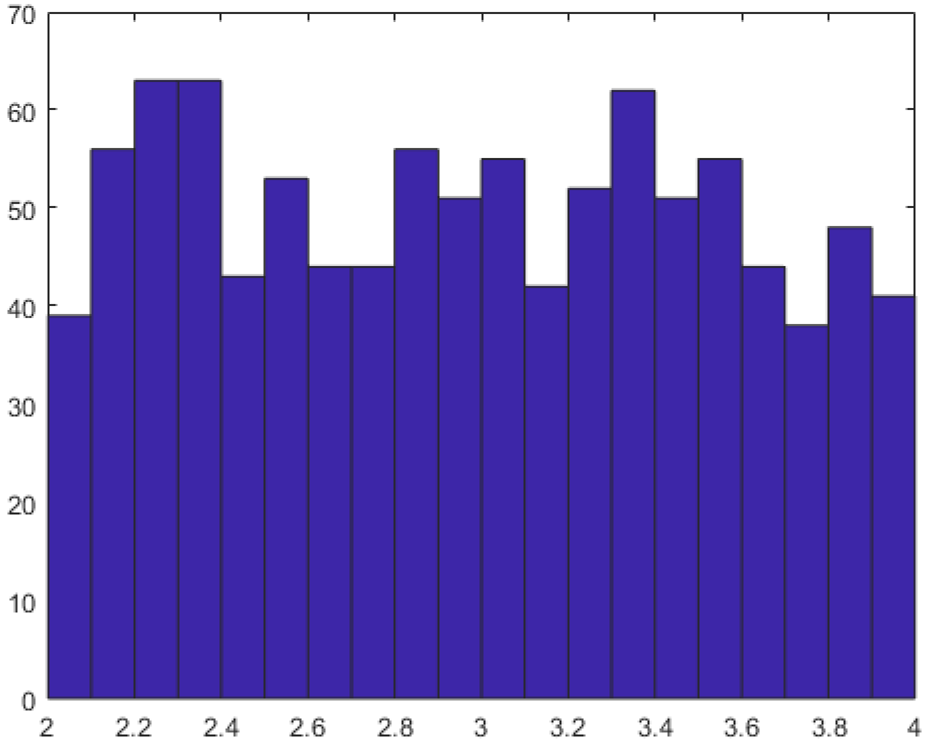
```
g = 
  Uncertain real parameter "g" with nominal value 2 and range [1.2,2.8].
```

```
g.Mode
```

```
ans = 
'Range'
```

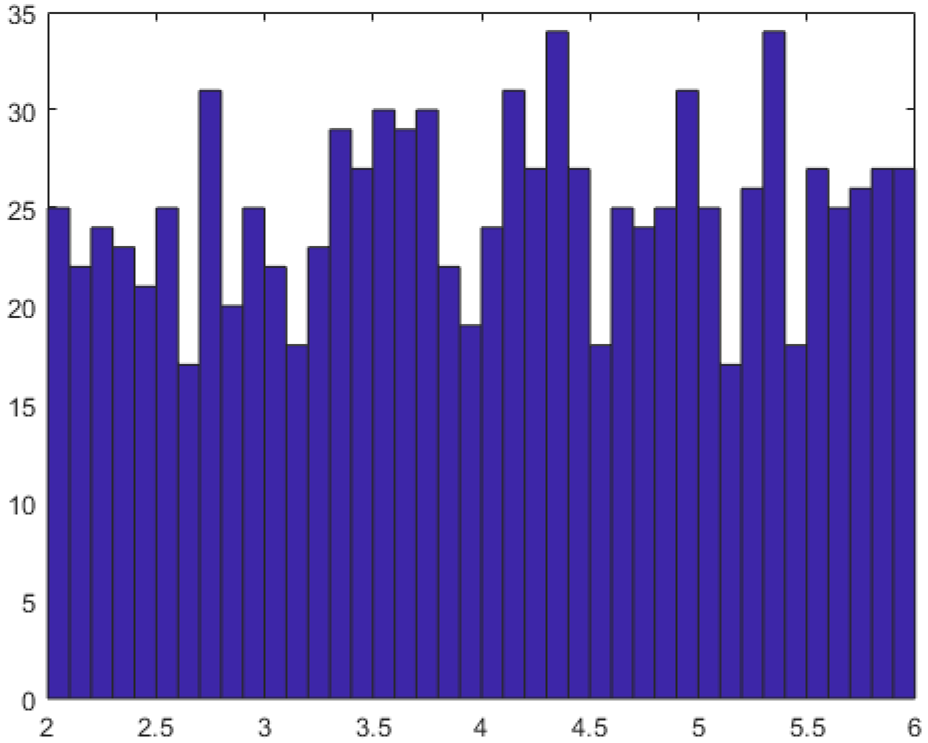
Create an uncertain real parameter, use `usample` to generate 1000 instances (resulting in a 1-by-1-by-1000 array), reshape the array, and plot a histogram, with 20 bins (within the range of 2 to 4).

```
h = ureal('h',3);
hsample = usample(h,1000);
hist(reshape(hsample,[1000 1]),20);
```



Make the range nonsymmetric about the nominal value, and repeat the sampling, and histogram plot (with 40 bins over the range of 2-to-6)

```
h.Range = [2 6];
hsample = usample(h,1000);
hist(reshape(hsample,[1000 1]),40);
```



The distribution is still uniform. The distribution used by `usample` is uniform in the actual value of the uncertain real parameter. However, highly skewed ranges can lead to poor numeric conditioning and poor results. Therefore, for meaningful results, avoid highly skewed ranges where the nominal value is orders of magnitude closer to one end of the range than to the other.

There is no notion of an empty `ureal` (or any other uncertain element, for that matter). `ureal`, by itself, creates an element named `'UNNAMED'`, with default property values.

See Also

`ureal`

Related Examples

- “System with Uncertain Parameters”
- “Uncertain LTI Dynamics Elements” on page 1-9

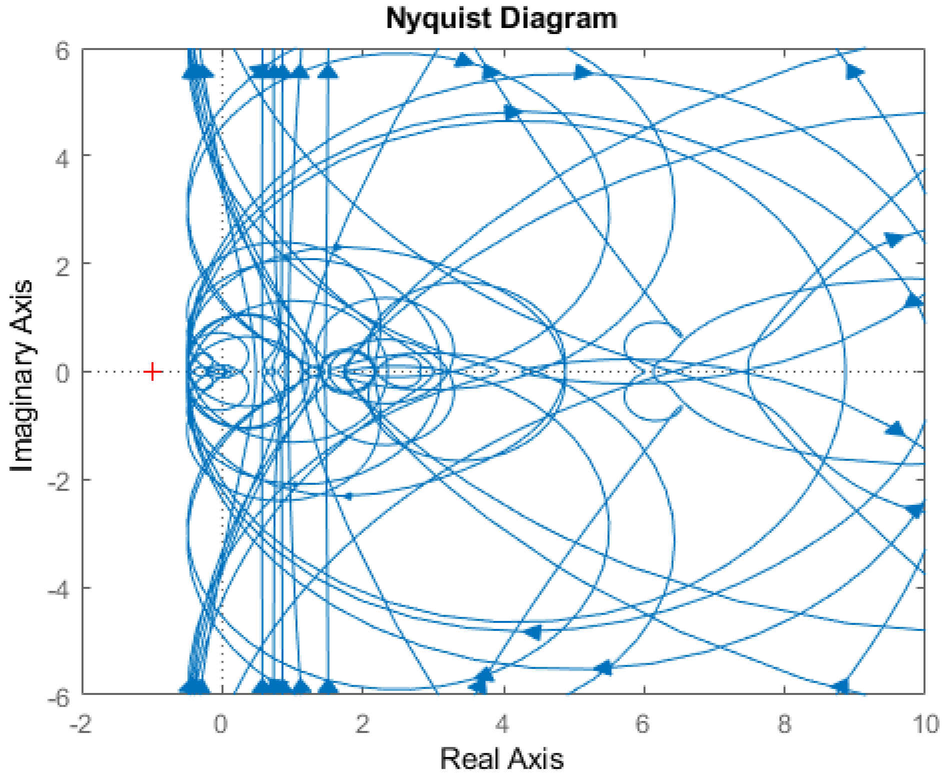
Uncertain LTI Dynamics Elements

Uncertain linear, time-invariant objects, ultidyn, are used to represent unknown linear, time-invariant dynamics, whose only known attributes are bounds on their frequency response.

Create Uncertain LTI Dynamics

You can create a 1-by-1 (scalar) positive-real uncertain linear dynamics element, whose frequency response always has real part greater than -0.5. Set the SampleStateDimension property to 5. Plot a Nyquist plot of 30 instances of the element.

```
g = ultidyn('g',[1 1],'Type','Positivereal','Bound',-0.5);
g.SampleStateDimension = 5;

nyquist(usample(g,30))
xlim([-2 10])
ylim([-6 6]);
```



Properties of ultidyn Elements

Uncertain linear, time-invariant objects have an internal name (the Name property), and are created by specifying their size (number of outputs and number of inputs).

The property Type specifies whether the known attributes about the frequency response are related to gain or phase. The property Type may be 'GainBounded' or 'PositiveReal'. The default value is 'GainBounded'.

The property Bound is a single number, which along with Type, completely specifies what is known about the uncertain frequency response. Specifically, if Δ is an ultidyn element, and if γ denotes the value of the Bound property, then the element represents the set of all stable, linear, time-invariant systems whose frequency response satisfies certain conditions:

If Type is 'GainBounded', $\dot{\bar{\sigma}}[\Delta(\omega)] \leq \gamma$ for all frequencies. When Type is 'GainBounded', the default value for Bound (i.e., γ) is 1. The NominalValue of Δ is always the 0-matrix.

If Type is 'PositiveReal', $\Delta(\omega) + \Delta^{*}(\omega) \geq 2\gamma\cdot$ for all frequencies. When Type is 'PositiveReal', the default value for Bound (i.e., γ) is 0. The NominalValue is always $(\gamma + 1 + 2|\gamma|)I$.

All properties of a ultidyn are accessible with get and set (although the NominalValue is determined from Type and Bound, and not accessible with set). The properties are

Properties	Meaning	Class
Name	Internal Name	char
NominalValue	Nominal value of element	See above
Type	'GainBounded' |'PositiveReal'	char
Bound	Norm bound or minimum real	scalar double
SampleStateDimension	State-space dimension of random samples of this uncertain element	scalar double
SampleMaxFrequency	Maximum natural frequency for random sampling	scalar double
AutoSimplify	'off' | {'basic'} |'full'	char

The SampleStateDim property specifies the state dimension of random samples of the element when using usample. The default value is 1. The AutoSimplify property serves the same function as in the uncertain real parameter.

Time Domain of ultidyn Elements

On its own, every ultidyn element is interpreted as a continuous-time, system with uncertain behavior, quantified by bounds (gain or real-part) on its frequency response. However, when a ultidyn element is an uncertain element of an uncertain state space model (uss), then the time-domain characteristic of the element is determined from the time-domain characteristic of the system. The bounds (gain-bounded or positivity) apply to the frequency-response of the element.

Interpreting Uncertainty in Discrete Time

The interpretation of a ultidyn element as a continuous-time or discrete-time system depends on the nature of the uncertain system (uss) within which it is an uncertain element.

For example, create a scalar ultidyn object. Then, create two 1-input, 1-output uss objects using the ultidyn object as their “D” matrix. In one case, create without specifying sample-time, which indicates continuous time. In the second case, force discrete-time, with a sample time of 0.42.

```
delta = ultidyn('delta',[1 1]);
sys1 = uss([],[],[],delta)
```

```
USS: 0 States, 1 Output, 1 Input, Continuous System
  delta: 1x1 LTI, max. gain = 1, 1 occurrence
sys2 = uss([],[],[],delta,0.42)
USS: 0 States, 1 Output, 1 Input, Discrete System, Ts = 0.42
  delta: 1x1 LTI, max. gain = 1, 1 occurrence
```

Next, get a random sample of each system. When obtaining random samples using `usample`, the values of the elements used in the sample are returned in the 2nd argument from `usample` as a structure.

```
[sys1s,d1v] = usample(sys1);
[sys2s,d2v] = usample(sys2);
```

Look at `d1v.delta.Ts` and `d2v.delta.Ts`. In the first case, since `sys1` is continuous-time, the system `d1v.delta` is continuous-time. In the second case, since `sys2` is discrete-time, with sample time 0.42, the system `d2v.delta` is discrete-time, with sample time 0.42.

```
d1v.delta.Ts
ans =
     0
d2v.delta.Ts
ans =
    0.4200
```

Finally, in the case of a discrete-time `uss` object, it is not the case that `ultidyn` objects are interpreted as continuous-time uncertainty in feedback with sampled-data systems. This very interesting hybrid theory is beyond the scope of the toolbox.

See Also

`ultidyn`

Related Examples

- "Uncertain Real Parameters" on page 1-4
- "Uncertain Matrices" on page 1-20
- "Uncertain State-Space Models" on page 1-27

Uncertain Gain and Phase

Use the `umargin` control design block to model gain and phase variations in feedback loops. Modeling gain and phase variations in your uncertain system model lets you verify stability margins during robustness analysis or enforce them during robust controller design.

To add gain and phase uncertainty to a feedback loop, you incorporate `umargin` blocks into an uncertain state-space (`uss`) model of the closed-loop system. `umargin` is a SISO control design block, representing gain and phase variation at a single location in a single feedback loop. To model gain and phase uncertainty in MIMO feedback systems, insert a separate `umargin` object at each location in the system at which you want to introduce gain and phase uncertainty.

Disk Model of Gain and Phase Uncertainty

`umargin` models gain and phase variations in an individual feedback channel as a frequency-dependent multiplicative factor $F(s)$ multiplying the nominal open-loop response $L(s)$, such that the perturbed response is $L(s)F(s)$. The factor $F(s)$ is parameterized by:

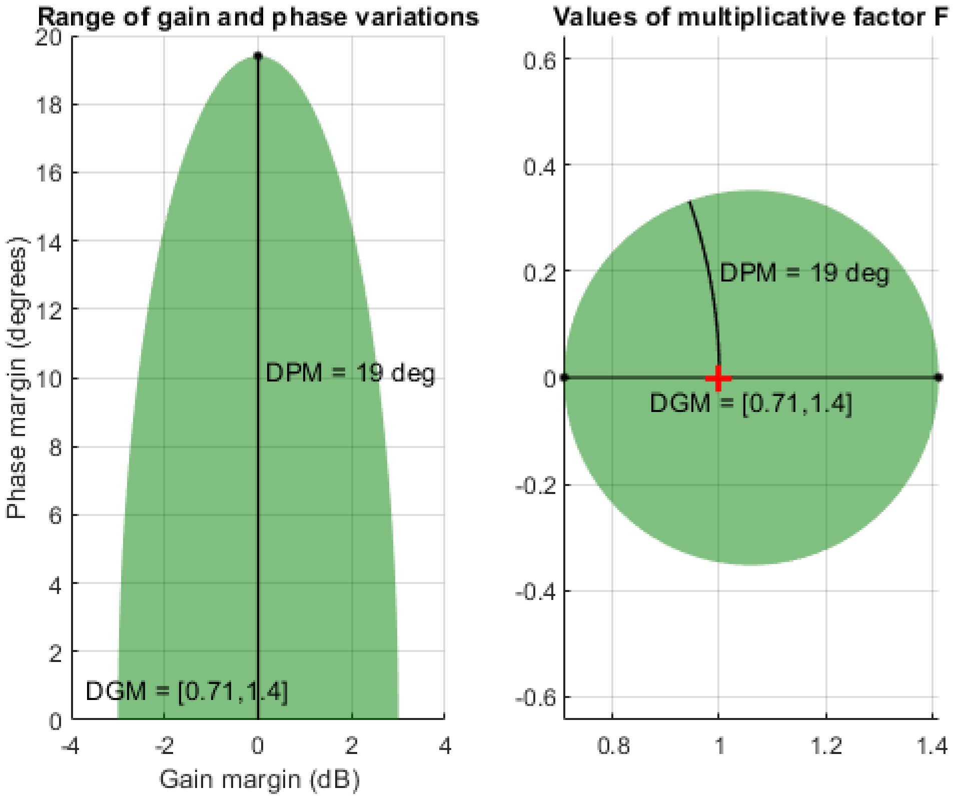
$$F(s) = \frac{1 + \alpha[(1 - E)/2]\delta(s)}{1 - \alpha[(1 + E)/2]\delta(s)}.$$

In this model,

- $\delta(s)$ is a gain-bounded dynamic uncertainty, normalized so that it always varies within the unit disk ($||\delta||_\infty < 1$).
- α sets the amount of gain and phase variation modeled by F. For fixed E, the parameter α controls the size of the disk. For $\alpha = 0$, the multiplicative factor is 1, corresponding to the nominal L.
- E, called the eccentricity, skews the modeled uncertainty toward gain increase or gain decrease.

The factor F takes values in a disk centered on the real axis and containing the nominal value $F = 1$. The disk is characterized by its intercept `DGM = [gmin,gmax]` with the real axis. `gmin` < 1 and `gmin` > 1 are the minimum and maximum relative changes in gain modeled by F, at nominal phase. The phase uncertainty modeled by F is the range `DPM = [pmin,pmax]` of phase values at the nominal gain ($|F| = 1$). For instance, in the following plot, the right side shows the disk F that intersects the real axis in the interval [0.71,1.4]. The left side shows that this disk models a gain variation of ±3 dB and a phase variation of ±19°.

```
F = umargin('F',1.4125)
plot(F)
```



When you create a `umargin` block, you specify the amount of uncertainty by specifying `DGM`. Use `getDGM` to translate specific amounts of gain and phase variations in to a suitable `DGM` range that captures these variations. For more information about the uncertainty model used by `umargin`, see "Stability Analysis Using Disk Margins" on page 2-2.

You can visualize the ranges of gain and phase uncertainty represented by a `umargin` object using `plot (umargin)`.

For examples of creating `umargin` objects and incorporating them into uncertain models, see:

- `umargin`
- "Model Gain and Phase Uncertainty in Feedback Loops" on page 1-31

Using Gain and Phase Uncertainty

When you have a `uss` model containing `umargin` control design blocks, you can perform robustness and worst-case analysis to examine how gain and phase variation affects the response of the system. For instance, use `robstab` and `robgain` to analyze the robust stability and robust performance of a system with gain and phase uncertainty. Use `wcgain` and `wcsigmaplot` to examine the worst-case responses of the system. For some examples, see:

- `umargin`
- "MIMO Stability Margins for Spinning Satellite" on page 2-14

Requiring robust stability for a closed-loop system with `umargin` gain and phase uncertainty is equivalent to enforcing a disk-based gain margin `[gmin,gmax]` and corresponding phase margin.

Therefore, you can use `umargin` blocks to enforce suitable disk margins when designing robust controllers with `musyn`. For examples, see:

- `umargin`
- “Robust Controller for Spinning Satellite” on page 3-102

The requirement that a closed-loop system is robust against a particular amount of gain and phase uncertainty is equivalent to saying that the system has that amount of gain and phase margin. You can therefore use a `umargin` block to check the disk-based stability margins of a system that also requires robustness against other types of uncertainty. For an example, see:

- “Check Robustness to Gain and Phase Variations” on the `umargin` reference page

See Also

`getDGM` | `plot (umargin)` | `umargin`

More About

- “Stability Analysis Using Disk Margins” on page 2-2
- “Model Gain and Phase Uncertainty in Feedback Loops” on page 1-31
- “MIMO Stability Margins for Spinning Satellite” on page 2-14

Uncertain Complex Parameters and Matrices

Uncertain Complex Parameters

The `ucomplex` element is the Control Design Block (Control System Toolbox) that represents an uncertain complex number. The value of an uncertain complex number lies in a disc, centered at `NominalValue`, with radius specified by the `Radius` property of the `ucomplex` element. The size of the disc can also be specified by `Percentage`, which means the radius is derived from the absolute value of the `NominalValue`. The properties of `ucomplex` objects are

Properties	Meaning	Class
`Name`	Internal Name	`char`
`NominalValue`	Nominal value of element	`double`
`Mode`	`'Range' | 'Percentage'`	`char`
`Radius`	Radius of disk	`double`
`Percentage`	Additive variation (percent of `Radius`)	`double`
`AutoSimplify`	`'off' | {'basic'} | 'full'`	`char`

The simplest construction requires only a name and nominal value. Displaying the properties shows that the default `Mode` is `Radius`, and the default radius is 1.

```
a = ucomplex('a',2-j)
```

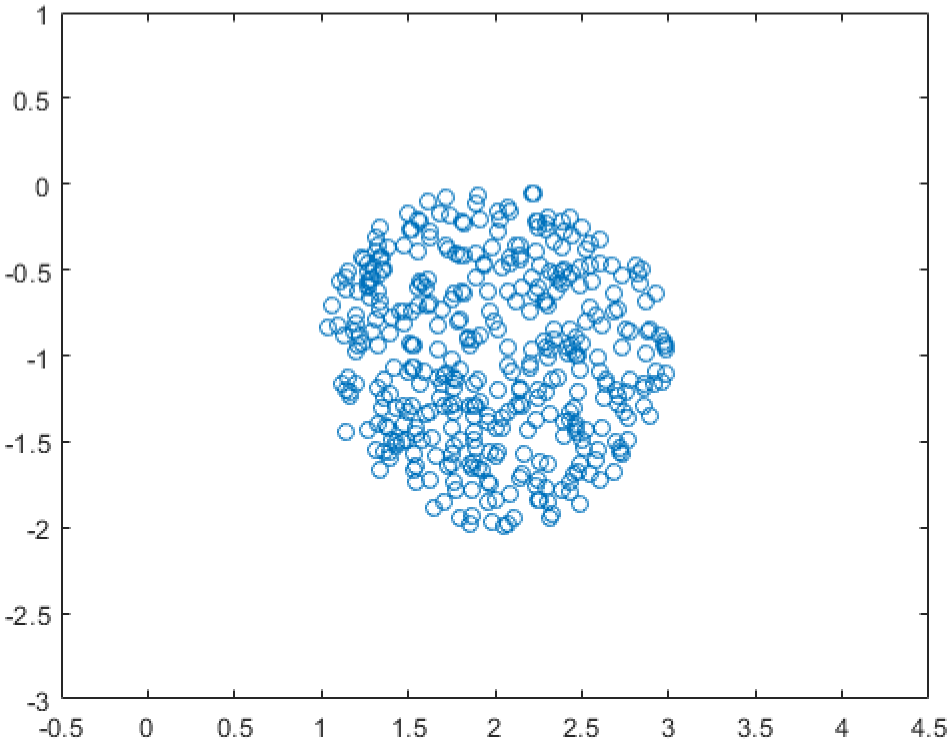
```
a =
  Uncertain complex parameter "a" with nominal value 2-1i and radius 1.
```

```
get(a)
```

```
    NominalValue: 2.0000 - 1.0000i
            Mode: 'Radius'
          Radius: 1
      Percentage: 44.7214
    AutoSimplify: 'basic'
            Name: 'a'
```

Sample the uncertain complex parameter at 400 values, and plot in the complex plane. Clearly, the samples appear to be from a disc of radius 1, centered in the complex plane at the value 2-*j*.

```
asample = usample(a,400);
plot(asample(:),'o');
xlim([-0.5 4.5]);
ylim([-3 1]);
```



Uncertain Complex Matrices

The uncertain complex matrix class, `ucomplexm`, represents the set of matrices given by the formula

$N + W_L \Delta W_R$

where N, W_L, and W_R are known matrices, and Δ is any complex matrix with $\dot{\bar{\sigma}}(\Delta) \le 1$. All properties of a `ucomplexm` are can be accessed with `get` and `set`. The properties are

Properties	Meaning	Class
`Name`	Internal Name	`char`
`NominalValue`	Nominal value of element	`double`
`WL`	Left weight	`double`
`WR`	Right weight	`double`
`AutoSimplify`	`'off' | {'basic'} | 'full'`	`char`

Uncertain Complex Matrix and Weighting Matrices

Create a 4-by-3 uncertain complex matrix (`ucomplexm`), and view its properties. The simplest construction requires only a name and nominal value.

```
m = ucomplexm('m',[1 2 3; 4 5 6; 7 8 9; 10 11 12])
```

```
m =
  Uncertain complex matrix "m" with 4 rows and 3 columns.
```

```
get(m)
```

```
    NominalValue: [4x3 double]
              WL: [4x4 double]
              WR: [3x3 double]
    AutoSimplify: 'basic'
            Name: 'm'
```

The nominal value is the matrix you supply to `ucomplexm`.

```
mnom = m.NominalValue
```

```
mnom = 4×3

     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

By default, the weighting matrices are the identity. For example, examine the left weighting.

```
m.WL
```

```
ans = 4×4

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Sample the uncertain matrix, and compare to the nominal value. Note the element-by-element sizes of the difference are roughly equal, indicative of the identity weighting matrices.

```
msamp = usample(m);
diff = abs(msamp-mnom)
```

```
diff = 4×3

    0.3309    0.0917    0.2881
    0.2421    0.3449    0.3917
    0.2855    0.2186    0.2915
    0.3260    0.2753    0.3816
```

Change the left and right weighting matrices, making the uncertainty larger as you move down the rows, and across the columns.

```
m.WL = diag([0.2 0.4 0.8 1.6]);
m.WR = diag([0.1 1 4]);
```

Sample the uncertain matrix again, and compare to the nominal value. Note the element-by-element sizes of the difference, and the general trend that the smallest differences are near the (1,1) element, and the largest differences are near the (4,3) element, consistent with the trend in the diagonal weighting matrices.

```
msamp = usample(m);
diff = abs(msamp-mnom)
```

```
diff = 4×3

    0.0048    0.0526    0.2735
    0.0154    0.1012    0.4898
    0.0288    0.3334    0.8555
    0.0201    0.4632    1.3783
```

See Also

ucomplex | ucomplexm

Systems with Unmodeled Dynamics

The unstructured uncertain dynamic system Control Design Block (Control System Toolbox), the udyn object, represents 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, multiplication (i.e., cascade) operate properly, and substitution (with usubs) is allowed. However, all of the analysis tools (e.g., robstab) do not handle these types of uncertain elements.

You can create a 2-by-3 udyn element. Check its size, and properties.

```
m = udyn('m',[2 3])
```

```
m =

  Uncertain dynamics "m" with 2 outputs and 3 inputs.
```

```
get(m)
```

```
     NominalValue: [2×3 ss]
    AutoSimplify: 'basic'
            Name: 'm'
              Ts: 0
        TimeUnit: 'seconds'
       InputName: {3×1 cell}
       InputUnit: {3×1 cell}
      InputGroup: [1×1 struct]
      OutputName: {2×1 cell}
      OutputUnit: {2×1 cell}
     OutputGroup: [1×1 struct]
           Notes: [0×1 string]
        UserData: []
```

See Also

udyn

Related Examples

- “Uncertain LTI Dynamics Elements” on page 1-9

Uncertain Matrices

Uncertain matrices (class umat) are built from doubles and uncertain elements, using traditional MATLAB matrix building syntax. Uncertain matrices can be added, subtracted, multiplied, inverted, transposed, etc., resulting in uncertain matrices. The rows and columns of an uncertain matrix are referenced in the same manner that MATLAB references rows and columns of an array, using parenthesis, and integer indices. The NominalValue of a uncertain matrix is the result obtained when all uncertain elements are replaced with their own NominalValue. The uncertain elements making up a umat are accessible through the Uncertainty gateway, and the properties of each element within a umat can be changed directly. The properties are:

Properties	Meaning	Class
NominalValue	Nominal value of element	double
Uncertainty	Uncertain blocks in the matrix, stored as a structure whose fields are named after the uncertain blocks, and contain the uncertain elements, such as ureal.	struct
SamplingGrid	Sampling grid, for umat arrays, stored as a structure whose fields are named after the sampling variables, and contain the sample values associated with the corresponding model in the array.	struct
Name	umat name. When you convert a static control design block such as ureal to an uncertain matrix using umat(blk), the Name property of the block is preserved.	char

Using usubs, specific values may be substituted for any of the uncertain elements within a umat. The command usample generates a random sample of the uncertain matrix, substituting random samples (within their ranges) for each of the uncertain elements.

The command wcnorm computes tight bounds on the worst-case (maximum over the uncertain elements' ranges) norm of the uncertain matrix.

Standard MATLAB numerical matrices (i.e., double) naturally can be viewed as uncertain matrices without any uncertainty.

Create and Manipulate Uncertain Matrices

You create uncertain matrices (umat objects) by creating uncertain parameters and using them to build matrices. You can then use uncertain matrices to build uncertain state-space models. This example shows how to create an uncertain matrix, access and change its uncertain parameters, extract elements, and perform matrix arithmetic.

For example, create two uncertain real parameters, and use them to create a 3-by-2 uncertain matrix.

```
a = ureal('a',3);
b = ureal('b',10,'Percentage',20);
M = [-a, 1/b; b, a+1/b; 1, 3]
```

```
M =
  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-1,1], 2 occurrences
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 3 occurrences

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

Examine and Modify umat Properties

M is a umat object. Examine its properties using get.

```
get(M)
```

```
    NominalValue: [3x2 double]
     Uncertainty: [1x1 struct]
    SamplingGrid: [1x1 struct]
            Name: ''
```

The nominal value of M is the matrix obtained by replacing all the uncertain elements with their nominal values.

```
M.NominalValue
```

```
ans = 3×2

   -3.0000    0.1000
   10.0000    3.1000
    1.0000    3.0000
```

The Uncertainty property is a structure containing the uncertain elements (the “Control Design Blocks” (Control System Toolbox)) of M.

```
M.Uncertainty
```

```
ans = struct with fields:
    a: [1x1 ureal]
    b: [1x1 ureal]
```

```
M.Uncertainty.a
```

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

Use the Uncertainty property for direct access to the uncertain elements. For example, check the Range of the uncertain element a within M.

```
M.Uncertainty.a.Range
```

```
ans = 1×2

     2     4
```

The range is [2,4] because you created the ureal parameter a with a nominal value 3 and the default uncertainty of +/- 1. Change the range to [2.5,5].

```
M.Uncertainty.a.Range = [2.5,5]
```

```
M =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-0.5,2], 2 occurrences
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 3 occurrences

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

This change to a only takes place within M. Verify that the variable a in the MATLAB workspace still has the original range.

```
a.Range
```

```
ans = 1×2

     2     4
```

You cannot combine elements that have a common internal name, but different properties. So, for example, entering M.Uncertainty.a - a would generate an error, because the realp parameter a in the workspace has different properties from the element a in M.

Row and Column Referencing

You can use standard row-column referencing to extract elements from a umat. For example, extract a 2-by-2 selection from M consisting of its second and third rows.

```
Msub = M(2:3,:)
```

```
Msub =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-0.5,2], 1 occurrences
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 2 occurrences

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

You can use single indexing only if the umat is a single column or row. Make a single-column selection from M and use single-index references to access elements of it.

```
Msing = M([2 1 2 3],2);
Msing(2)
```

```
ans =

  Uncertain matrix with 1 rows and 1 columns.
  The uncertainty consists of the following blocks:
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 1 occurrences

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

You can use indexing to change the value of any element of a umat. For example, set the (3,2) entry of M to an uncertain parameter c.

```
c = ureal('c',3,'Percentage',40);
M(3,2) = c
```

```
M =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-0.5,2], 2 occurrences
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 2 occurrences
    c: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

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

M now has three uncertain blocks.

Matrix Operations on umat Objects

You can perform many matrix operations on a `umat` object, such as matrix-multiply, transpose, and inverse. You can also combine uncertain matrices with numeric matrices that do not have uncertainty.

For example, premultiply `M` by a `1-by-3` numeric matrix, resulting in a 1-by-2 `umat`.

```
M1 = [2 3 1]*M;
```

Verify that the first entry of `M1` is as expected, `-2*a + 3*b + 1`.

```
d = M1(1) - (-2*M.Uncertainty.a + 3*M.Uncertainty.b + 1)
```

```
d =

  Uncertain matrix with 1 rows, 1 columns, and no uncertain blocks.

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

Transpose `M`, form a product, and invert it. As expected, the product of a matrix and its inverse is the identity matrix. You can verify this by sampling the result.

```
H = M.'*M;
K = inv(H);
usample(K*H,3)
```

```
ans =
ans(:,:,1) =

    1.0000    0.0000
   -0.0000    1.0000


ans(:,:,2) =

    1.0000    0.0000
   -0.0000    1.0000


ans(:,:,3) =

    1.0000    0.0000
   -0.0000    1.0000
```

Lifting a Double Matrix to umat

You can convert a numeric matrix to a `umat` object with no uncertain elements. Use the `umat` command to *lift* a double matrix to the `umat` class. For example:

```
Md = [1 2 3;4 5 6];
M = umat(Md)
```

```
M =

  Uncertain matrix with 2 rows, 3 columns, and no uncertain blocks.

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

You can also convert higher-dimension numeric matrices to `umat`. When you do so, the software interprets the third dimension and beyond as array dimensions. For example, convert a random three-dimensional numeric array to `umat`.

```
Md = randn(4,5,6);
M = umat(Md)
```

```
M =

  6x1 array of uncertain matrices with 4 rows, 5 columns, and no uncertain blocks.

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

The result is a one-dimensional array of uncertain matrices, rather than a three-dimensional uncertain array. Similarly, a four-dimensional numeric array converts to a two-dimensional array of `umat` objects.

```
Md = randn(4,5,6,7);
M = umat(Md)
```

```
M =

  6x7 array of uncertain matrices with 4 rows, 5 columns, and no uncertain blocks.

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

See “Array Management for Uncertain Objects” on page 1-58 for more information about multidimensional arrays of uncertain objects.

See Also
`umat` | `ureal`

Related Examples

- “Uncertain State-Space Models” on page 1-27

Evaluate Uncertain Elements by Substitution

You can make substitutions for uncertain elements in uncertain matrices and models using `usubs`. Doing so is useful for evaluating uncertain objects at particular values of the uncertain parameters, or for sampling uncertain objects at multiple parameter values.

For example, create an uncertain matrix with three uncertain parameters.

```
a = ureal('a',3);
b = ureal('b',10,'Percentage',20);
c = ureal('c',3,'Percentage',40);
M = [-a, 1/b; b, a+1/b; 1, c]
```

```
M =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-1,1], 2 occurrences
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 3 occurrences
    c: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

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

Substitute all instances of the uncertain real parameter `a` with the value 4. This operation results in a `umat` containing only two uncertain real parameters, `b` and `c`.

```
M2 = usubs(M,'a',4)
```

```
M2 =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    b: Uncertain real, nominal = 10, variability = [-20,20]%, 3 occurrences
    c: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "M2.NominalValue" to see the nominal value, "get(M2)" to see all properties, and "M2.Uncerta
```

You can replace all instances of one uncertain real parameter with another. For example, replace all instances of `b` in `M` with the uncertain parameter `a`. The resulting `umat` contains only the parameters `a` and `c`, and has two additional occurrences of `a`, compared to `M`.

```
M3 = usubs(M,'b',M.Uncertainty.a)
```

```
M3 =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 3, variability = [-1,1], 5 occurrences
    c: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "M3.NominalValue" to see the nominal value, "get(M3)" to see all properties, and "M3.Uncerta
```

Next, evaluate `M` at the nominal value of `a` and a random value of `b`.

```
M4 = usubs(M,'a','NominalValue','b','Random')
```

```
M4 =
  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    c: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "M4.NominalValue" to see the nominal value, "get(M4)" to see all properties, and "M4.Uncerta
```

Use the `usample` command to generate multiple random instances of `umat`, `uss`, or `ufrd` uncertain objects. See “Generate Samples of Uncertain Systems” on page 1-50 for more information.

See Also

`ufrd` | `umat` | `usample` | `uss` | `usubs`

Related Examples

- “Substitution by usubs” on page 1-55
- “Generate Samples of Uncertain Systems” on page 1-50

Uncertain State-Space Models

Uncertain state-space (uss) models are linear systems with uncertain state-space matrices and/or uncertain linear dynamics. Like their numeric (i.e., not uncertain) counterpart, the ss model object, you can build them from state-space matrices using the ss command. When one or more of the state-space matrices contain uncertain elements (uncertain Control Design Blocks (Control System Toolbox)), the result is a uss model object.

Combining uncertain systems with other uncertain systems (for example, using model arithmetic, connect, or feedback) usually results in an uncertain system. You can also combine numeric systems with uncertain systems. Usually the result is an uncertain system. The nominal value of an uncertain system is a ss model object.

In the example below, the A, B and C matrices are made up of uncertain real parameters. Packing them together with the ss command results in a continuous-time uncertain system.

Uncertain State-Space Model

To create an uncertain state-space model, you first use Control Design Blocks to create uncertain elements. Then, use the elements to specify the state-space matrices of the system.

For instance, create three uncertain real parameters and build state-spaces matrices from them.

```
p1 = ureal('p1',10,'Percentage',50);
p2 = ureal('p2',3,'PlusMinus',[-.5 1.2]);
p3 = ureal('p3',0);

A = [-p1 p2; 0 -p1];
B = [-p2; p2+p3];
C = [1 0; 1 1-p3];
D = [0; 0];
```

The matrices constructed with uncertain parameters, A, B, and C, are uncertain matrix (umat) objects. Using them as inputs to ss results in a 2-output, 1-input, 2-state uncertain system.

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

```
sys =

  Uncertain continuous-time state-space model with 2 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p1: Uncertain real, nominal = 10, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

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

The display shows that the system includes the three uncertain parameters.

Properties of uss Objects

uss models, like all model objects, include properties that store dynamics and model metadata. View the properties of an uncertain state-space model.

```
p1 = ureal('p1',10,'Percentage',50);
p2 = ureal('p2',3,'PlusMinus',[-.5 1.2]);
p3 = ureal('p3',0);
A = [-p1 p2; 0 -p1];
B = [-p2; p2+p3];
C = [1 0; 1 1-p3];
D = [0; 0];
sys = ss(A,B,C,D);     % create uss model

get(sys)
```

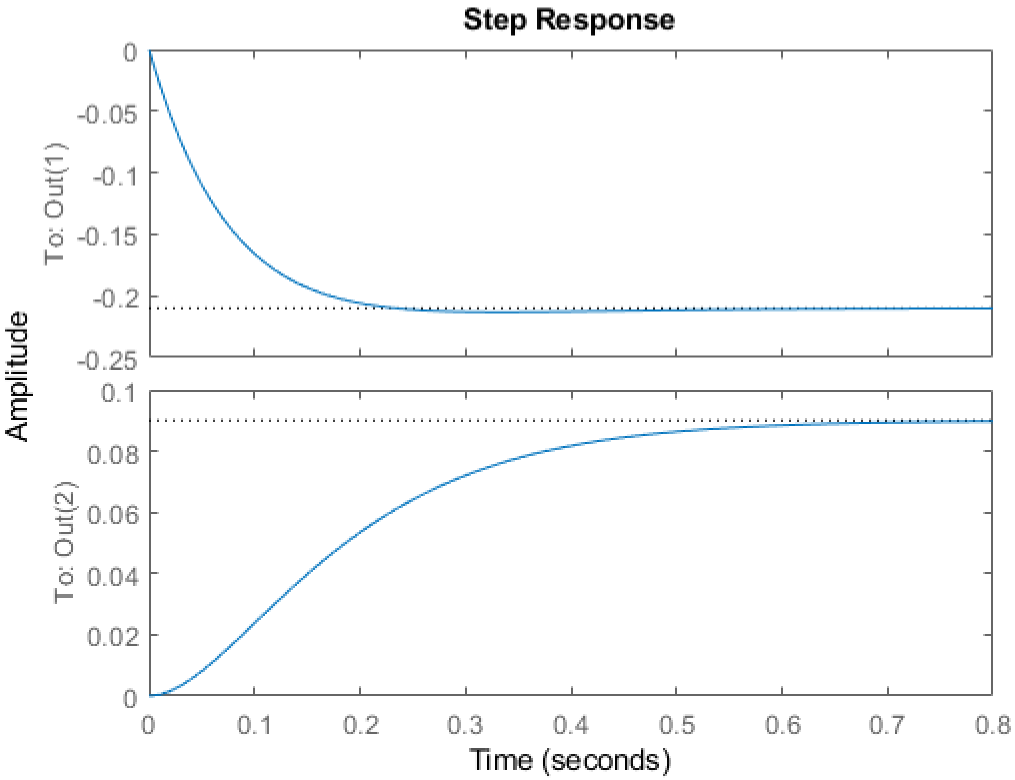
```
     NominalValue: [2x1 ss]
      Uncertainty: [1x1 struct]
                A: [2x2 umat]
                B: [2x1 umat]
                C: [2x2 umat]
                D: [2x1 double]
                E: []
        StateName: {2x1 cell}
        StateUnit: {2x1 cell}
    InternalDelay: [0x1 double]
       InputDelay: 0
      OutputDelay: [2x1 double]
               Ts: 0
         TimeUnit: 'seconds'
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1x1 struct]
       OutputName: {2x1 cell}
       OutputUnit: {2x1 cell}
      OutputGroup: [1x1 struct]
            Notes: [0x1 string]
         UserData: []
             Name: ''
     SamplingGrid: [1x1 struct]
```

Most of the properties behave similarly to how they behave for `ss` model objects. The `NominalValue` property is itself an `ss` model object. You can therefore analyze the nominal value as you would any state-space model. For instance, compute the poles and step response of the nominal system.

```
pole(sys.NominalValue)
```

```
ans = 2×1

   -10
   -10
```

```
step(sys.NominalValue)
```



As with the uncertain matrices (`umat`), the `Uncertainty` property is a structure containing the uncertain elements. You can use this property for direct access to the uncertain elements. For instance, check the `Range` of the uncertain element named `p2` within `sys`.

```
sys.Uncertainty.p2.Range
```

```
ans = 1×2

    2.5000    4.2000
```

Change the uncertainty range of `p2` within `sys`.

```
sys.Uncertainty.p2.Range = [2 4];
```

This command changes only the range of the parameter called `p2` in `sys`. It does not change the variable `p2` in the MATLAB workspace.

```
p2.Range
```

```
ans = 1×2

    2.5000    4.2000
```

Lifting a ss to a uss

A not-uncertain state space object may be interpreted as an uncertain state space object that has no dependence on uncertain elements. Use the uss command to “lift” a ss to the uss class.

```
sys = rss(3,2,1);
usys = uss(sys)
USS: 3 States, 2 Outputs, 1 Input, Continuous System
```

Arrays of ss objects can also be lifted. See “Array Management for Uncertain Objects” on page 1-58 for more information about how arrays of uncertain objects are handled.

See Also

ss | umat | uss

Related Examples

- “Create and Manipulate Uncertain Matrices” on page 1-20
- “Uncertain Model Interconnections” on page 1-41

Model Gain and Phase Uncertainty in Feedback Loops

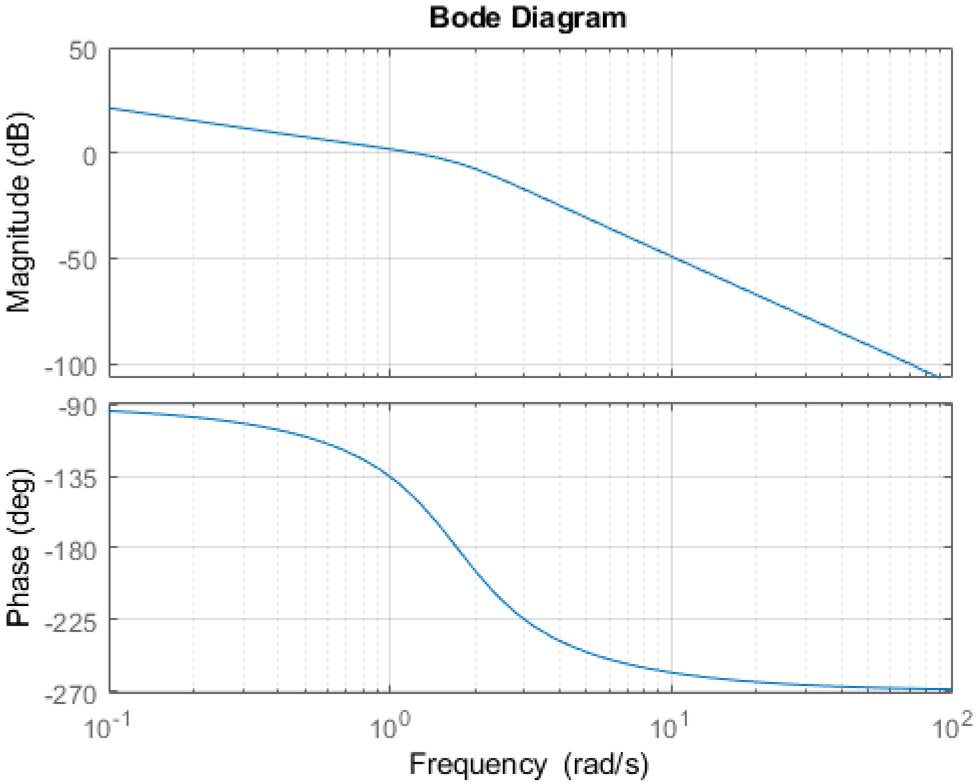
This example shows how to model gain and phase uncertainty in feedback loops using the `umargin` control design block. The example also shows how to check a feedback loop for robust stability against such uncertainty.

Modeling Gain and Phase Uncertainty

Consider a SISO feedback loop with open-loop transfer function

$$L = \frac{3.5}{s^3 + 2s^2 + 3s}.$$

```
L = tf(3.5,[1 2 3 0]);
bode(L)
grid on
```



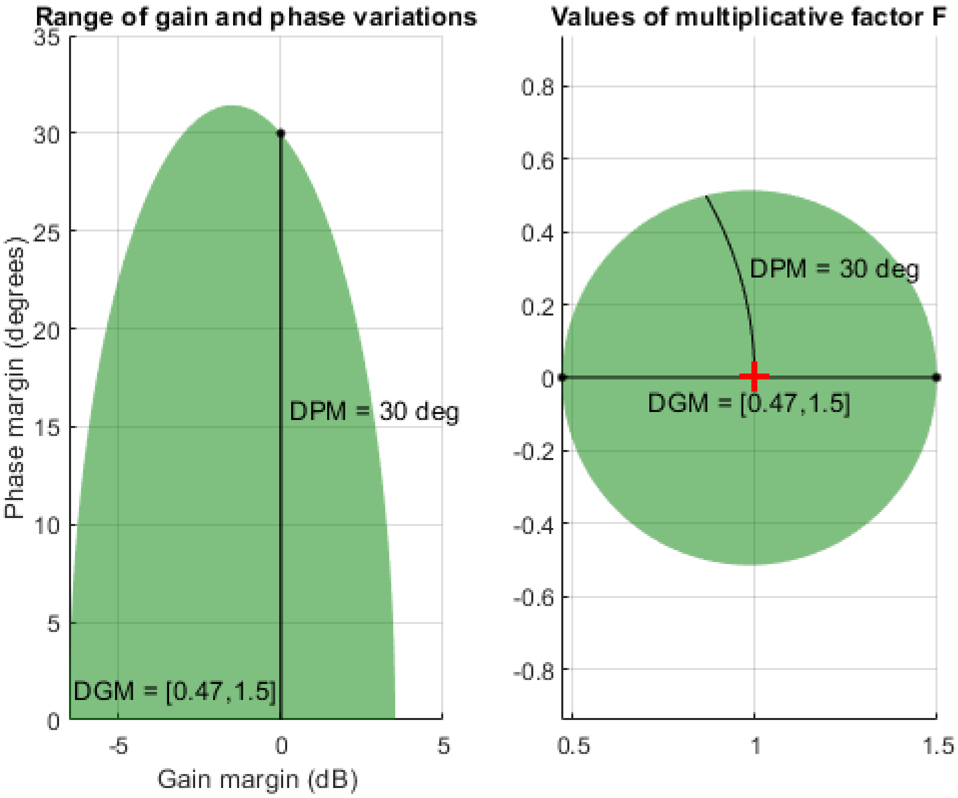
Due to plant uncertainty and other sources of variability, the loop gain and phase are subject to fluctuations. In general, you can quantify the amount of uncertainty through experimenting on your system, or approximate it based on insight or experience. For this example, suppose that the open-loop gain can increase or decrease by 50%, and the phase by ±30°. You can use the `umargin` block to model such uncertainty. `umargin` represents the variation as an uncertain multiplicative factor F with nominal value 1. The set of values F can take captures the gain and phase uncertainty you specify.

To create the `umargin` block, use `getDGM` to compute the smallest uncertainty disk that captures the gain and phase variation you want to represent. Use the output of `getDGM` to create F.

```
DGM = getDGM(1.5,30,'tight');
F = umargin('F',DGM)

F = 
  Uncertain gain/phase "F" with relative gain change in [0.472,1.5] and phase change of ±30 degre
```

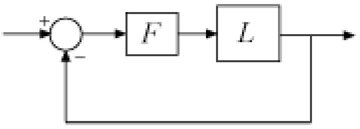
Visualize *F* to see the range of values taken by this factor (right) and the range of gain and phase variations it models by F (left).

```
plot(F)
```



The plots show that the gain can vary between 47% and 150% of its nominal value (assuming no phase variation) and the phase can vary by ±30° (assuming no gain variation). When both gain and phase vary, their variation stays inside the shaded region in the left plot.

The uncertainty F multiplies the open-loop response, yielding a closed-loop system as in the following diagram.



Incorporate this uncertainty into the closed-loop model.

```
T = feedback(L*F,1)
```

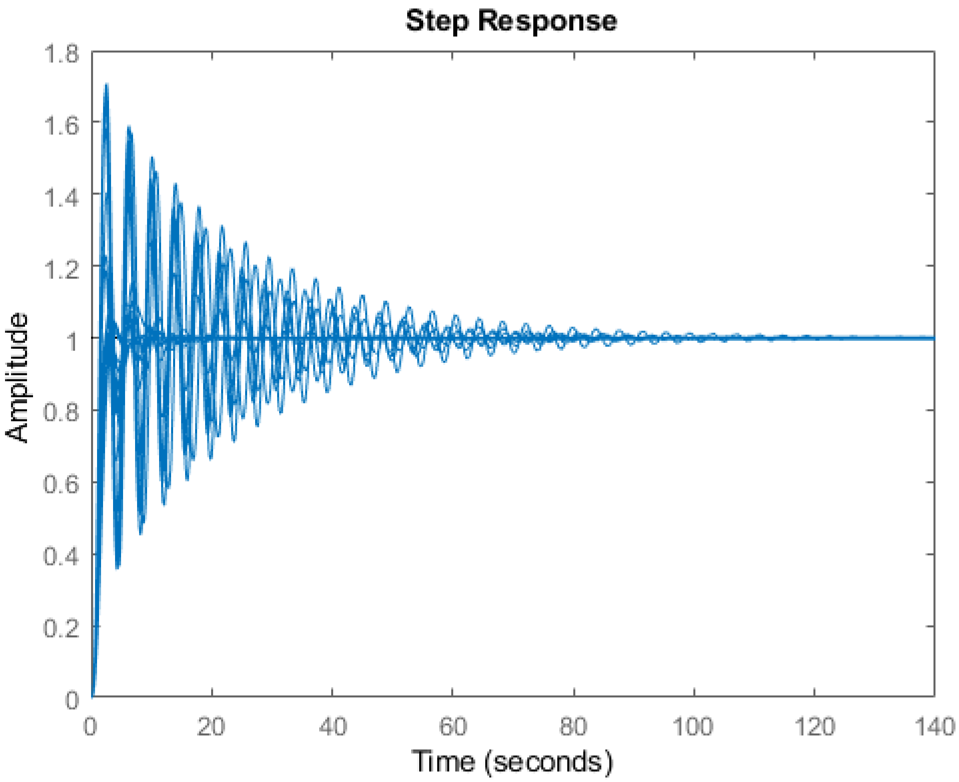
```
T =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 3 states.
  The model uncertainty consists of the following blocks:
    F: Uncertain gain/phase, gain × [0.472,1.5], phase ± 30 deg, 1 occurrences

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

The result is an uncertain state-space (`uss`) model of the closed-loop system containing the uncertain block *F*. In general the open-loop gain can contain other uncertain blocks too.

Robustness Analysis

Sampling the uncertainty and plotting the closed-loop step response suggest poor robustness to such gain/phase variations.

```
clf
rng default
step(T)
```



To quantify this poor robustness, use `robstab` to gauge the robust stability margin for the specified uncertainty.

```
SM = robstab(T)
```

```
SM = struct with fields:
          LowerBound: 0.8303
```

```
           UpperBound: 0.8319
    CriticalFrequency: 1.4482
```

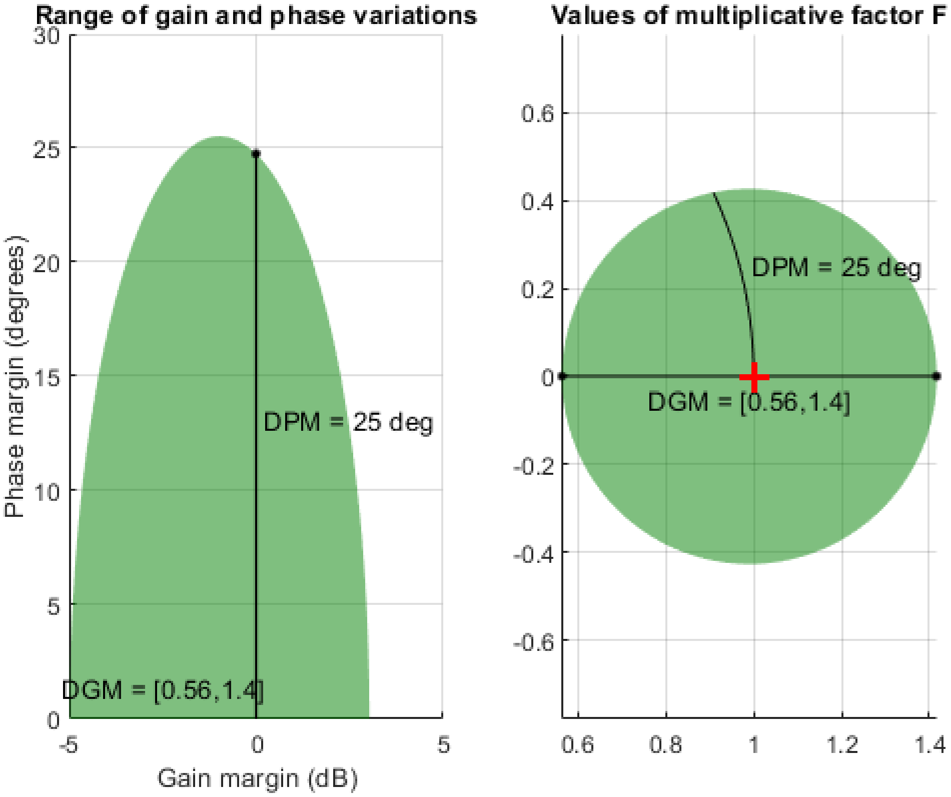
The robust stability margin is only 0.83, meaning that the feedback loop can only withstand 83% of the specified uncertainty. The factor 0.83 is in normalized units. To translate this value into an actual safe range of gain and phase variations, use `uscale`. This command takes a modeled uncertainty disk and a scaling factor, and converts it into a new uncertainty disk.

```
Fsafe = uscale(F,0.83)
```

```
Fsafe =
  Uncertain gain/phase "F" with relative gain change in [0.564,1.42] and phase change of ±24.8 de
```

The display shows that 83% of the uncertainty specified in `F` (and therefore in `L`) amounts to gain variation between 56% and 142% of the nominal value, and phase variation of ±25°. Plot the disk `Fsafe` to see the full range of simultaneous gain and phase variations that the closed-loop system can tolerate.

```
plot(Fsafe)
```



In the model `L`, gain and phase uncertainty is the only source of uncertainty. Therefore, you can obtain the same result by directly computing the disk-based margins with `diskmargin`. Make sure to account for the "eccentricity" of the uncertainty model `F`, which biases the uncertainty toward gain increase or decrease.

```
E = F.Eccentricity;
DM = diskmargin(L,E)
```

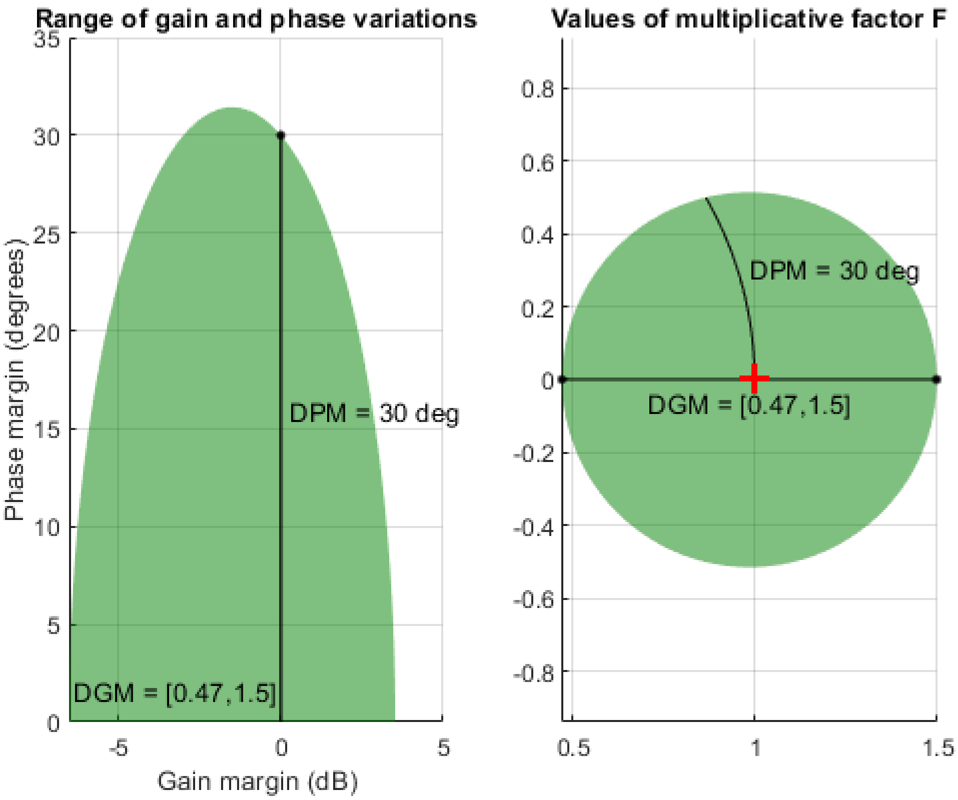
```
DM = struct with fields:
           GainMargin: [0.5626 1.4178]
          PhaseMargin: [-24.8091 24.8091]
           DiskMargin: 0.4274
           LowerBound: 0.4274
           UpperBound: 0.4274
            Frequency: 1.4505
    WorstPerturbation: [1x1 ss]
```

This returns the disk-based gain and phase margins for the feedback loop L. These values coincide with the ranges displayed for the scaled uncertainty Fsafe.

Choice of Eccentricity

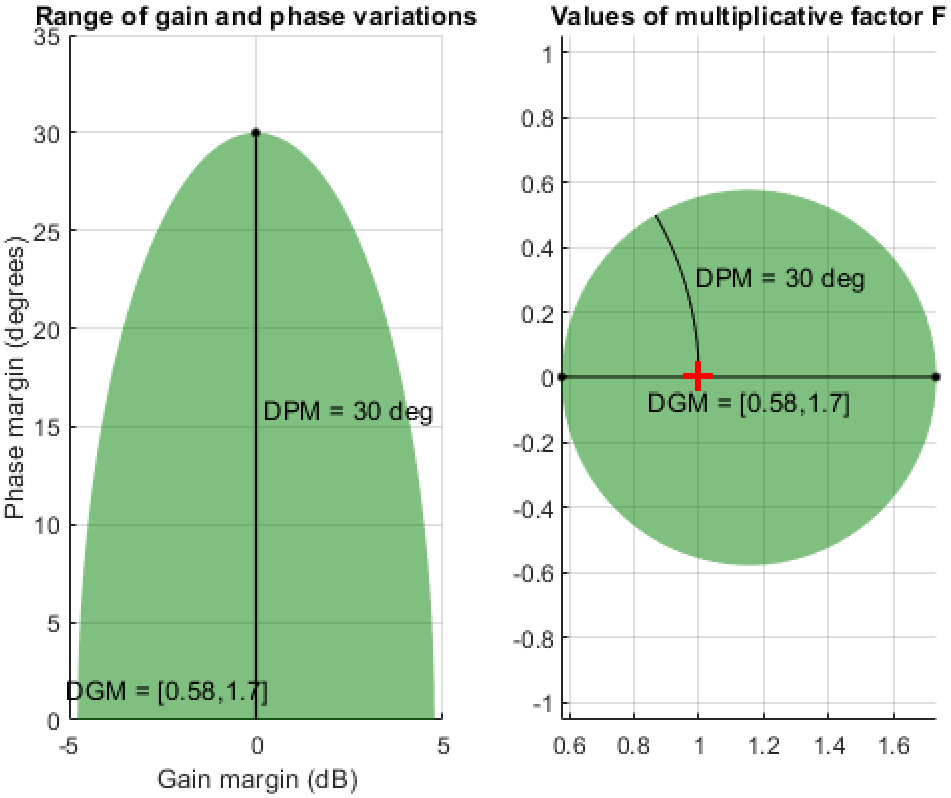
In the calculations above, you used getDGM to map ±50% gain and ±30° phase uncertainty into the disk of uncertainty F. You used the 'tight' option, which picks the smallest disk that captures both the specified gain and phase uncertainty. Examining the range of gain and variations encompassed by *F* again shows that the gain range is skewed toward gain decrease.

```
plot(F)
```



Alternatively, you can use the 'balanced' option of getDGM to use a model with equal amounts of (relative) gain increase and decrease. The balanced range corresponds to zero eccentricity (E=0) in diskmargin.

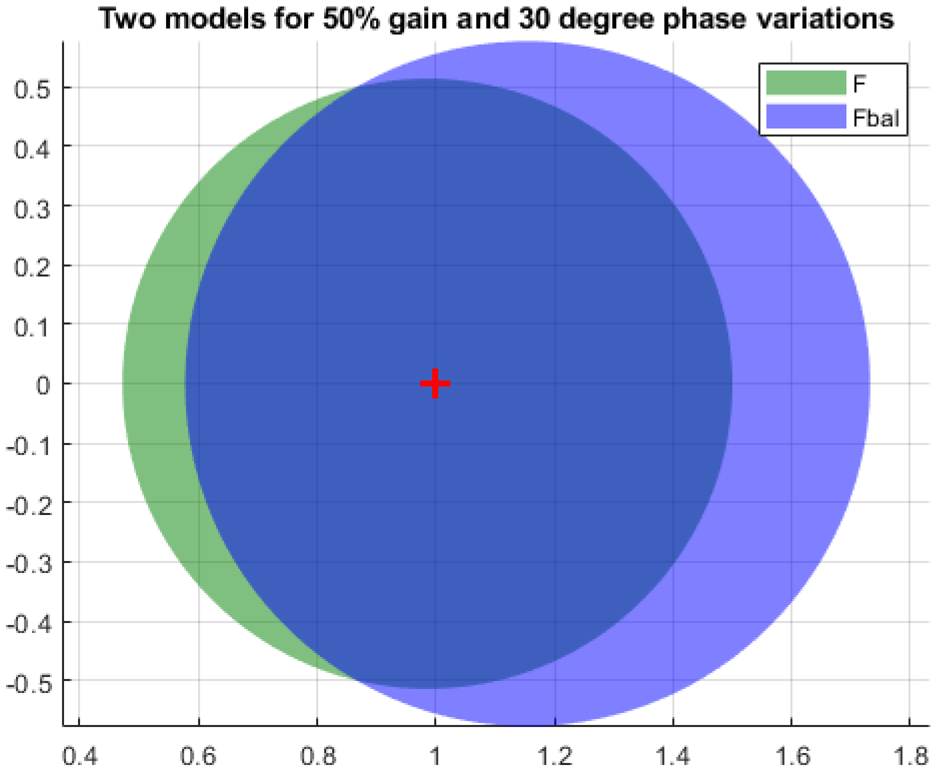
```
DGM = getDGM(1.5,30,'balanced');
Fbal = umargin('Fbal',DGM);
plot(Fbal)
```



This time the gain range shown in the left plot is symmetric.

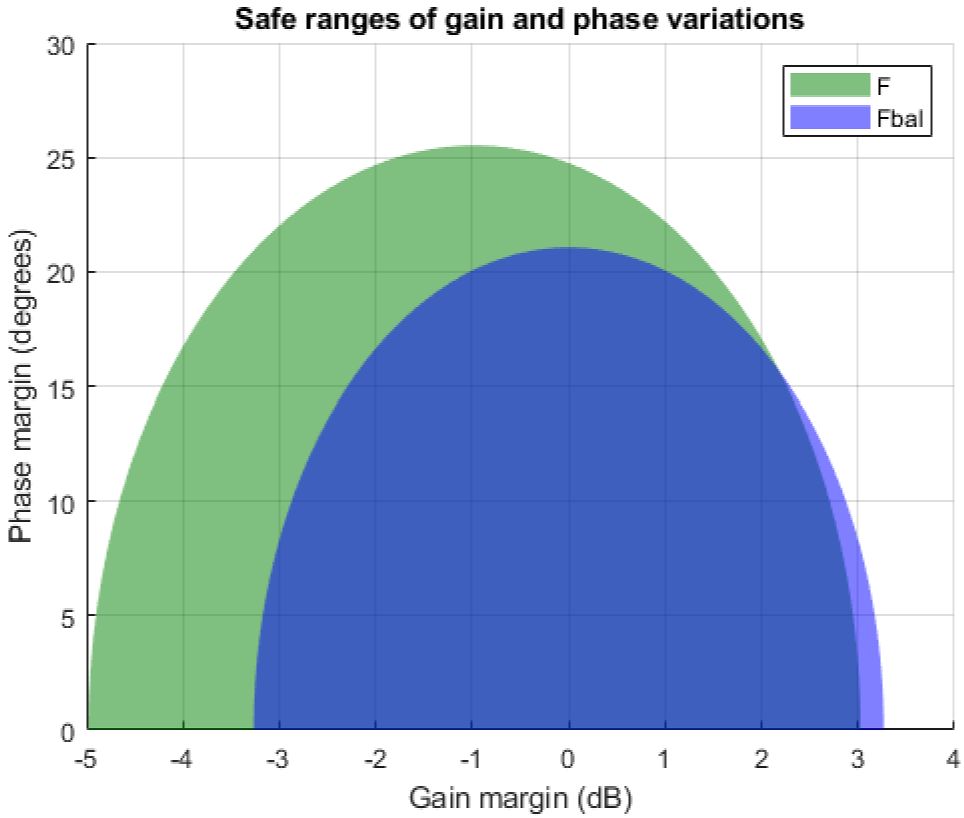
Next, compare the disk of values for the two uncertainty models F and Fbal. The uncertainty disk is larger for the 'balanced' option.

```
clf
DGM = F.GainChange;
DGMbal = Fbal.GainChange;
diskmarginplot([DGM;DGMbal],'disk')
legend('F','Fbal')
title('Two models for 50% gain and 30 degree phase variations')
```



Now compute the robust stability margin for the system with `Fbal` and compare the safe ranges of gain and phase variations for the two models.

```
SM2 = robstab(feedback(L*Fbal,1));
Fbalsafe = uscale(Fbal,SM2.LowerBound);

DGMsafe = Fsafe.GainChange;
DGMbalsafe = Fbalsafe.GainChange;
diskmarginplot([DGMsafe;DGMbalsafe])
legend('F','Fbal')
title('Safe ranges of gain and phase variations')
```



The `'tight'` fit `F` yields a larger safe region and gets closer to the original robustness target (3.5 dB gain margin and 30 degrees phase margin).

See Also

`diskmarginplot` | `getDGM` | `plot (umargin)` | `umargin`

More About

- “Uncertain Gain and Phase” on page 1-12
- “Stability Analysis Using Disk Margins” on page 2-2

Sample Uncertain Systems

The command `usample` randomly samples the uncertain system at a specified number of points. Randomly sample an uncertain system at 20 points in its modeled uncertainty range. This gives a 20-by-1 `ss` array. Consequently, all analysis tools from Control System Toolbox™ are available.

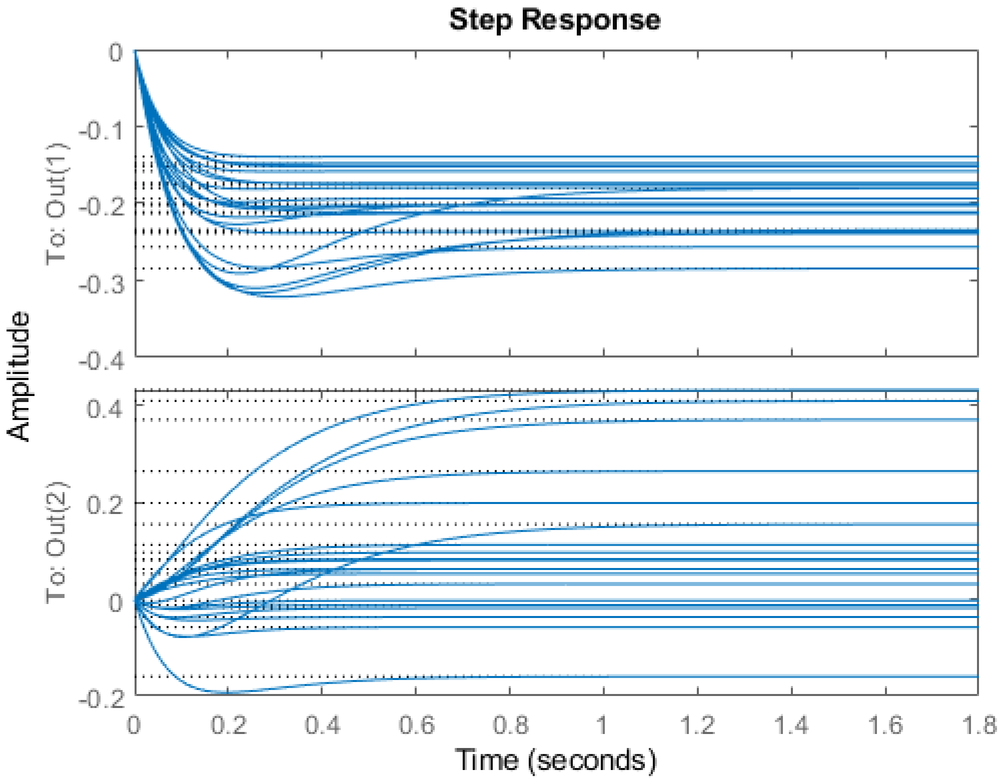
```
p1 = ureal('p1',10,'Percentage',50);
p2 = ureal('p2',3,'PlusMinus',[-.5 1.2]);
p3 = ureal('p3',0);
A = [-p1 p2; 0 -p1];
B = [-p2; p2+p3];
C = [1 0; 1 1-p3];
D = [0; 0];

sys = ss(A,B,C,D) % Create uncertain state-space model
```

```
sys =

  Uncertain continuous-time state-space model with 2 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p1: Uncertain real, nominal = 10, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

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

```
manysys = usample(sys,20);
size(manysys)
```

```
20x1 array of state-space models.
Each model has 2 outputs, 1 inputs, and 2 states.
```

```
stepplot(manysys)
```



The command `stepplot` can be called directly on a `uss` object. The default behavior samples the `uss` object at 20 instances, and plots the step responses of these 20 models, as well as the nominal value.

The same features are available for other analysis commands such as `bodeplot`, `bodemag`, `impulse`, and `nyquist`.

See Also

`usample` | `uss`

Related Examples

- "Generate Samples of Uncertain Systems" on page 1-50

Uncertain Model Interconnections

Feedback Around an Uncertain Plant

It is possible to form interconnections of uss objects. A common example is to form the feedback interconnection of a given controller with an uncertain plant.

First create the uncertain plant. Start with two uncertain real parameters.

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

Next, create an unmodeled dynamics element, delta, and a first-order weighting function, whose DC value is 0.2, high-frequency gain is 10, and whose crossover frequency is 8 rad/sec.

```
delta = ultidyn('delta',[1 1],'SampleStateDimension',5);
W = makeweight(0.2,6,6);
```

Finally, create the uncertain plant consisting of the uncertain parameters and the unmodeled dynamics.

```
P = tf(gamma,[tau 1])*(1+W*delta);
```

You can create an integral controller based on nominal plant parameters. Nominally the closed-loop system will have damping ratio of 0.707 and time constant of 2*tau.

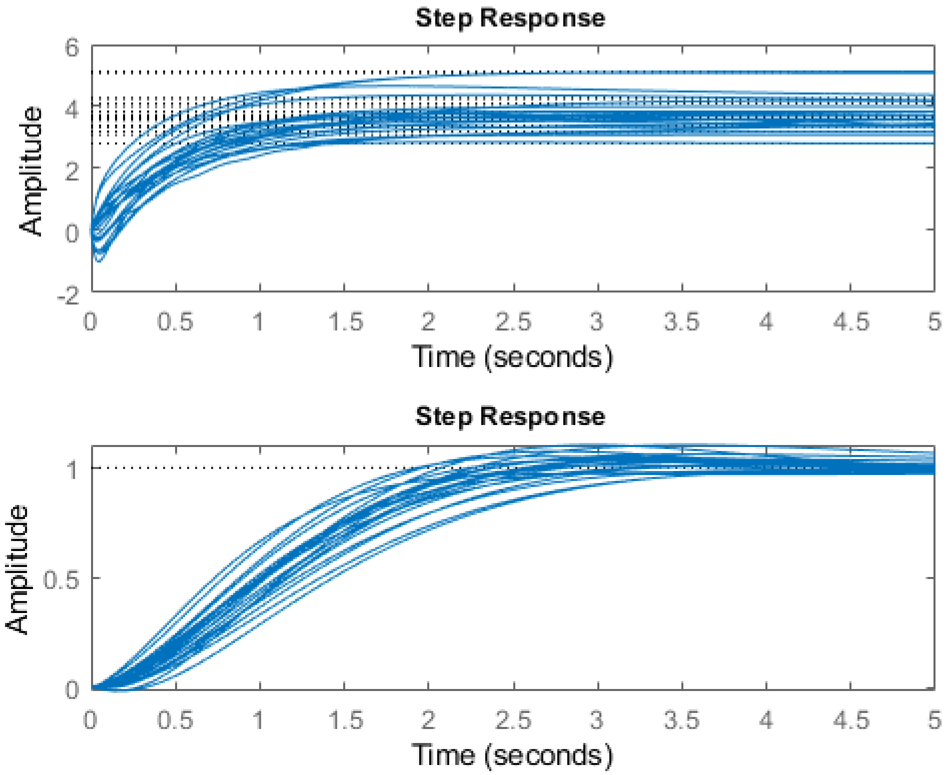
```
KI = 1/(2*tau.Nominal*gamma.Nominal);
C = tf(KI,[1 0]);
```

Create the uncertain closed-loop system using the feedback command.

```
CLP = feedback(P*C,1);
```

Plot samples of the open-loop and closed-loop step responses. As expected the integral controller reduces the variability in the low frequency response.

```
subplot(2,1,1);
stepplot(P,5)
subplot(2,1,2);
stepplot(CLP,5)
```



Basic Model Interconnections

All the model arithmetic and model-interconnection commands of Control System Toolbox software work with uncertain models. These include:

- `connect`
- `feedback`
- `series`
- `parallel`
- `append`
- `blkdiag`
- `lft`
- `stack`

For more information about model interconnections, see “Model Interconnection” (Control System Toolbox).

Create Uncertain Frequency Response Data Models

Uncertain frequency responses (`ufrd`) arise naturally when computing the frequency response of an uncertain state-space model (`uss`). They also arise when frequency response data in an `frd` model object is combined with an uncertain matrix (`umat`) such as by adding, multiplying, or concatenating.

To take the frequency response of an uncertain state-space model, use the `ufrd` command. Construct an uncertain state-space model.

```
p1 = ureal('p1',10,'pe',50);
p2 = ureal('p2',3,'plusm',[-.5 1.2]);
p3 = ureal('p3',0);
A = [-p1 p2;0 -p1];
B = [-p2;p2+p3];
C = [1 0;1 1-p3];
D = [0;0];
sys = ss(A,B,C,D)
```

```
sys =

  Uncertain continuous-time state-space model with 2 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p1: Uncertain real, nominal = 10, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

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

Compute the uncertain frequency response of the uncertain system. Use `ufrd` command with a frequency grid of 100 points. The result is an uncertain frequency response model object, a `ufrd` model.

```
sysg = ufrd(sys,logspace(-2,2,100))
```

```
sysg =

  Uncertain continuous-time FRD model with 2 outputs, 1 inputs, 100 frequency points.
    p1: Uncertain real, nominal = 10, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

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

Properties of `ufrd` Model Objects

View the properties of the model object.

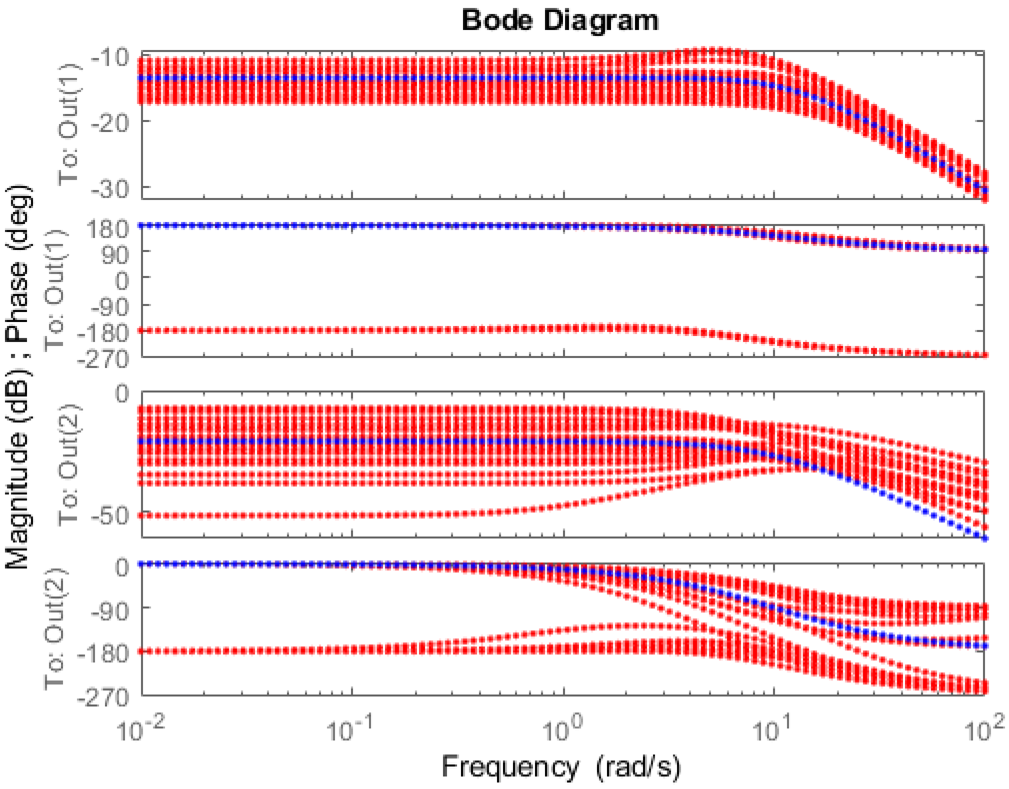
```
get(sysg)
```

```
        Frequency: [100x1 double]
    FrequencyUnit: 'rad/TimeUnit'
     ResponseData: [2x1x100x1 umat]
     NominalValue: [2x1 frd]
      Uncertainty: [1x1 struct]
       InputDelay: 0
      OutputDelay: [2x1 double]
               Ts: 0
         TimeUnit: 'seconds'
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1x1 struct]
       OutputName: {2x1 cell}
       OutputUnit: {2x1 cell}
      OutputGroup: [1x1 struct]
            Notes: [0x1 string]
         UserData: []
             Name: ''
     SamplingGrid: [1x1 struct]
```

The properties ResponseData and Frequency behave the same as the corresponding properties in Control System Toolbox™ frd objects, except that ResponseData is an uncertain matrix (umat). The properties InputName, OutputName, InputGroup, and OutputGroup behave in exactly the same manner as for all of the Control System Toolbox model objects such as ss, zpk, tf, and frd.

The NominalValue property is an frd object. Hence all functions you can use to analyze frd objects can also analyze ufrd objects. are available. When you use analysis commands such as bode or step with an uncertain model, the command plots random samples of the response to give you a sense of the variation. For instance, plot sampled Bode responses of the system along with the nominal response, using a dot marker so that you can see the individual frequency points.

```
bode(sysg,'r.',sysg.NominalValue,'b.')
```

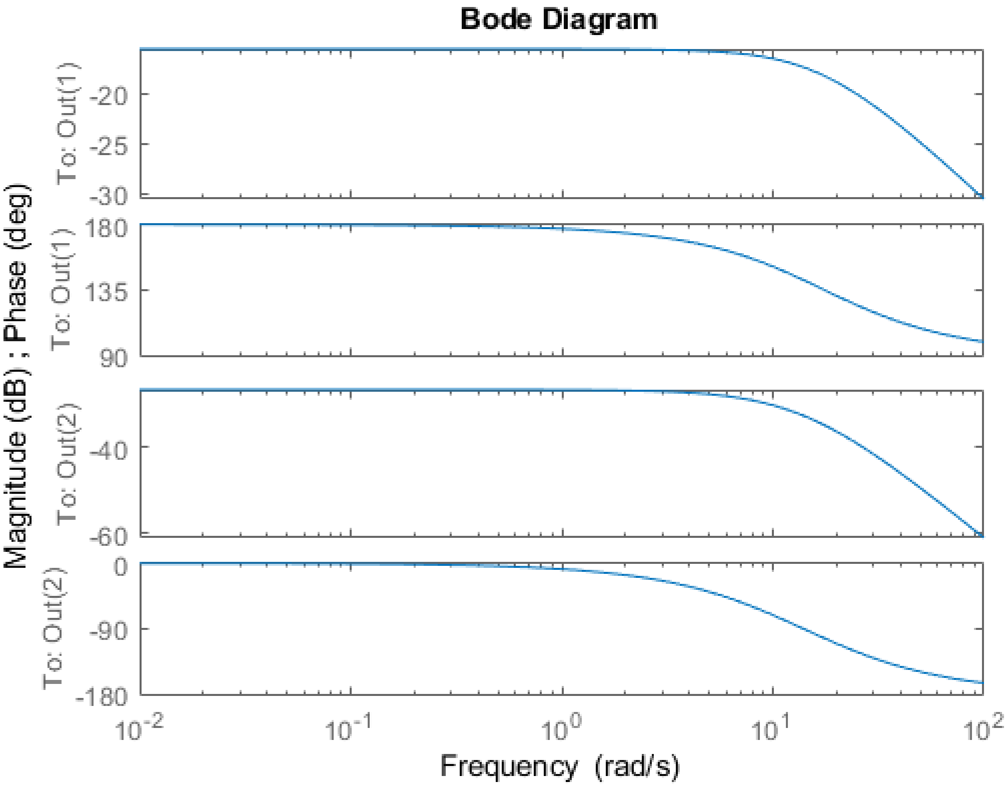


Just as with umat uncertain matrices and uss uncertain models, the Uncertainty property of the ufrd model is a structure containing the uncertain elements. In the model sysg, all uncertain elements are `ureal` parameters. Change the nominal value of the uncertain element `p1` within `sysg` to 14, and plot the Bode response of the (new) nominal system.

```
sysg.Uncertainty.p1.NominalValue = 14
```

```
sysg =

  Uncertain continuous-time FRD model with 2 outputs, 1 inputs, 100 frequency points.
    p1: Uncertain real, nominal = 14, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

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

```
bode(sysg.NominalValue)
```



Lifting an `frd` model to a `ufrd` model

A non-uncertain frequency response model is equivalent to an uncertain frequency response model with no uncertain elements. Use the `ufrd` command to “lift” an `frd` model to the `ufrd` class.

```
sys = rss(3,2,1);
sysg = frd(sys,logspace(-2,2,100));
usysg = ufrd(sysg)
```

```
usysg =

  Uncertain continuous-time FRD model with 2 outputs, 1 inputs, 100 frequency points, and no unce
```

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

You can also lift arrays of `frd` objects. See “Array Management for Uncertain Objects” on page 1-58 for more information about how arrays of uncertain objects are handled.

See Also

`ufrd`

Simplifying Representation of Uncertain Objects

A minimal realization of the transfer function matrix

$$H(s) = \begin{bmatrix} \frac{2}{s+1} & \frac{4}{s+1} \\ \frac{3}{s+1} & \frac{6}{s+1} \end{bmatrix}$$

has only 1 state, obvious from the decomposition

$$H(s) = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \frac{1}{s+1} [1 \ \ 2].$$

However, a “natural” construction, formed by

```
sys11 = ss(tf(2,[1 1]));
sys12 = ss(tf(4,[1 1]));
sys21 = ss(tf(3,[1 1]));
sys22 = ss(tf(6,[1 1]));
sys = [sys11 sys12;sys21 sys22]
a =
       x1  x2  x3  x4
   x1  -1   0   0   0
   x2   0  -1   0   0
   x3   0   0  -1   0
   x4   0   0   0  -1
b =
       u1  u2
   x1   2   0
   x2   0   2
   x3   2   0
   x4   0   2
c =
        x1   x2   x3   x4
   y1    1    2    0    0
   y2    0    0  1.5    3
d =
       u1  u2
   y1   0   0
   y2   0   0
Continuous-time model
```

has four states, and is nonminimal.

In the same manner, the internal representation of uncertain objects built up from uncertain elements can become nonminimal, depending on the sequence of operations in their construction. The command `simplify` employs ad-hoc simplification and reduction schemes to reduce the complexity of the representation of uncertain objects. There are three levels of simplification: off, basic and full. Each uncertain element has an `AutoSimplify` property whose value is either `'off'`, `'basic'` or `'full'`. The default value is `'basic'`.

After (nearly) every operation, the command `simplify` is automatically run on the uncertain object, cycling through all of the uncertain elements, and attempting to simplify (without error) the representation of the effect of that uncertain object. The `AutoSimplify` property of each element dictates the types of computations that are performed. In the `'off'` case, no simplification is even attempted. In 'basic', fairly simple schemes to detect and eliminate nonminimal representations are used. Finally, in 'full', numerical based methods similar to truncated balanced realizations are used, with a very tight tolerance to minimize error.

Effect of the Autosimplify Property

Create an uncertain real parameter, view the AutoSimplify property of a, and then create a 1-by-2 umat, both of whose entries involve the uncertain parameter.

```
a = ureal('a',4);
a.AutoSimplify
ans =
basic
m1 = [a+4 6*a]
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 1 occurrence
```

Note that although the uncertain real parameter a appears in both (two) entries of the matrix, the resulting uncertain matrix m1 only depends on “1 occurrence” of a.

Set the AutoSimplify property of a to 'off' (from 'basic'). Recreate the 1-by-2 umat. Now note that the resulting uncertain matrix m2 depends on “2 occurrences” of a.

```
a.AutoSimplify = 'off';
m2 = [a+4 6*a]
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 2 occurrences
```

The 'basic' level of autosimplification often detects (and simplifies) duplication created by linear terms in the various entries. Higher order (quadratic, bilinear, etc.) duplication is often not detected by the 'basic' autosimplify level.

For example, reset the AutoSimplify property of a to 'basic' (from 'off'). Create an uncertain real parameter, and a 1-by-2 umat, both of whose entries involve the square of the uncertain parameter.

```
a.AutoSimplify = 'basic';
m3 = [a*(a+4) 6*a*a]
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 4 occurrences
```

Note that the resulting uncertain matrix m3 depends on “4 occurrences” of a.

Set the AutoSimplify property of a to 'full' (from 'basic'). Recreate the 1-by-2 umat. Now note that the resulting uncertain matrix m4 depends on “2 occurrences” of a.

```
a.AutoSimplify = 'full';
m4 = [a*(a+4) 6*a*a]
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 2 occurrences
```

Although m4 has a less complex representation (2 occurrences of a rather than 4 as in m3), some numerical variations are seen when both uncertain objects are evaluated at (say) 0.

```
usubs(m3,'a',0)
ans =
     0     0
usubs(m4,'a',0)
ans =
  1.0e-015 *
   -0.4441         0
```

Small numerical differences are also noted at other evaluation points. The example below shows the differences encountered evaluating at a equal to 1.

```
usubs(m3,'a',1)
ans =
     5     6
usubs(m4,'a',1)
ans =
    5.0000    6.0000
```

Direct Use of simplify

The simplify command can be used to override all uncertain element's AutoSimplify property. The first input to the simplify command is an uncertain object. The second input is the desired reduction technique, which can either 'basic' or 'full'.

Again create an uncertain real parameter, and a 1-by-2 umat, both of whose entries involve the square of the uncertain parameter. Set the AutoSimplify property of a to 'basic'.

```
a.AutoSimplify = 'basic';
m3 = [a*(a+4) 6*a*a]
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 4 occurrences
```

Note that the resulting uncertain matrix m3 depends on four occurrences of a.

The simplify command can be used to perform a 'full' reduction on the resulting umat.

```
m4 = simplify(m3,'full')
UMAT: 1 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1  1], 2 occurrences
```

The resulting uncertain matrix m4 depends on only two occurrences of a after the reduction.

See Also

simplify

Related Examples

- “Introduction to Uncertain Elements” on page 1-2
- “Decomposing Uncertain Objects” on page 1-71

Generate Samples of Uncertain Systems

Use the usample function to randomly sample an uncertain model, returning one or more non-uncertain instances of the uncertain model.

Generating One Sample

If A is an uncertain object, then usample(A) generates a single sample of A.

For example, a sample of a ureal is a scalar double.

```
A = ureal('A',6);
B = usample(A)
B =
    5.7298
```

Create a 1-by-3 umat with A and an uncertain complex parameter C. A single sample of this umat is a 1-by-3 double.

```
C = ucomplex('C',2+6j);
M = [A C A*A];
usample(M)
ans =
   5.9785             1.4375 + 6.0290i  35.7428
```

Generating Many Samples

If A is an uncertain object, then usample(A,N) generates N samples of A.

For example, 20 samples of a ureal gives a 1-by-1-20 double array.

```
B = usample(A,20);
size(B)
ans =
     1     1    20
```

Similarly, 30 samples of the 1-by-3 umat M yields a 1-by-3-by-30 array.

```
size(usample(M,30))
ans =
     1     3    30
```

See “Sample Uncertain Elements to Create Arrays” on page 1-64 for more information on sampling uncertain objects.

Sampling Uncertain LTI Dynamics

When sampling an ultidyn element or an uncertain object that contains a ultidyn element, the result is always a state-space (ss) object. The property SampleStateDimension of the ultidyn class determines the state dimension of the samples. The same is true when sampling umargin objects, since these are a type of dynamic uncertainty.

Create a 1-by-1, gain bounded ultidyn object with gain bound 4. Verify that the default state dimension for samples is 3.

```
del = ultidyn('del',[1 1],'Bound',4);
del.SampleStateDimension
```

```
ans = 3
```

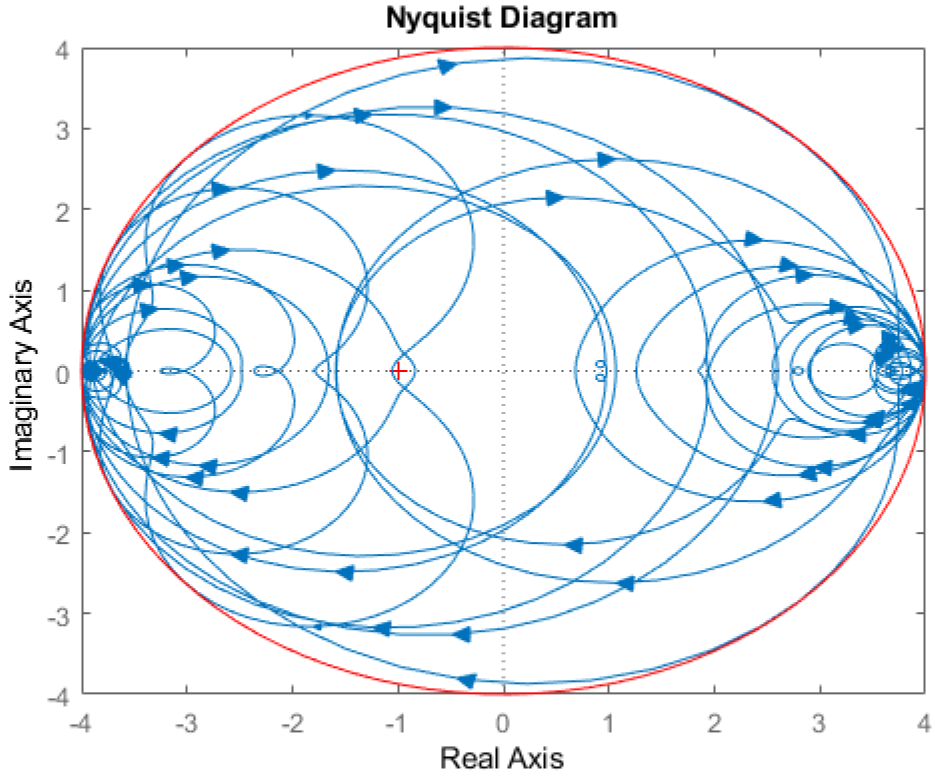
Sample the uncertain element at 30 points. Verify that this creates a 30-by-1 `ss` array of 1-input, 1-output, 1-state systems.

```
rng(0)  % for reproducibility
delS = usample(del,30);
size(delS)
```

```
30x1 array of state-space models.
Each model has 1 outputs, 1 inputs, and 3 states.
```

Plot the Nyquist plot of these samples and add a disk of radius 4, the gain bound of `del`.

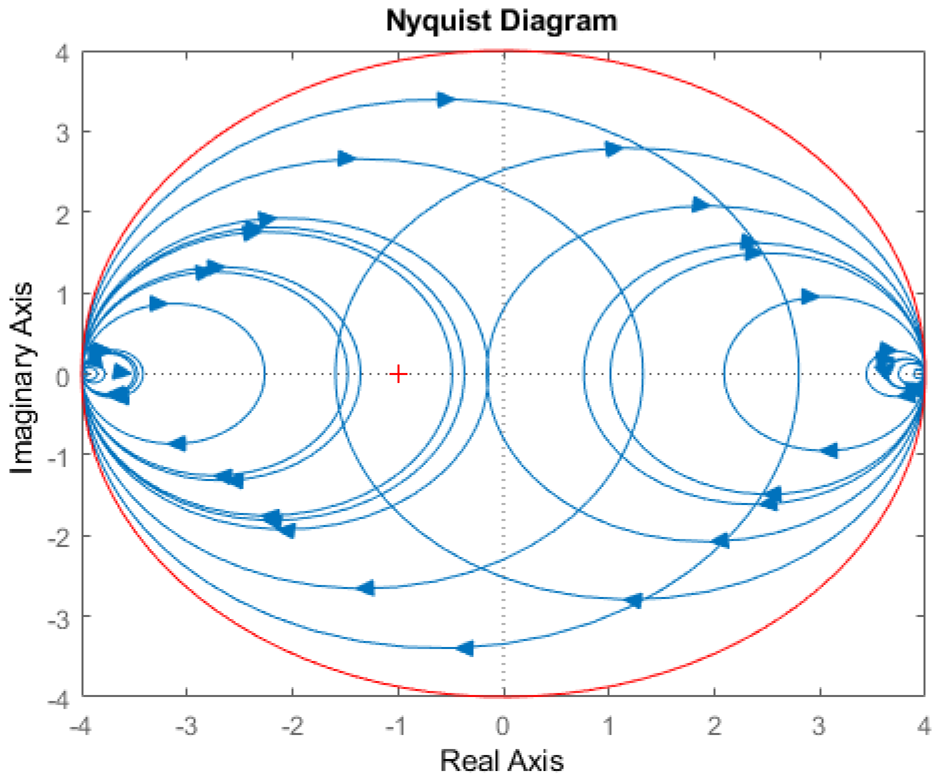
```
nyquist(delS)
hold on;
theta = linspace(-pi,pi);
plot(del.Bound*exp(sqrt(-1)*theta),'r');
hold off;
```



Change `SampleStateDimension` to 1, and repeat entire procedure. The Nyquist plots again satisfy the gain bound, but the Nyquist plots are all circles, indicative of 1st order systems.

```
del.SampleStateDimension = 1;
delS = usample(del,30);
```

```
nyquist(delS)
hold on;
theta = linspace(-pi,pi);
plot(del.Bound*exp(sqrt(-1)*theta),'r');
hold off;
```



With SampleStateDimension = 1, all Nyquist plots touch the gain boundary at either (-1,0) or (1,0) (frequency = 0 or Inf). Higher sampling dimension yields a Nyquist curve that reaches the gain bound at more frequencies, yielding more thorough coverage.

Create a umargin object using the default SampleStateDimension. The umargin block models uncertain gain and phase. The modeled variations are within bounded ranges. For this example use a umargin block that captures relative gain variations of a factor of two in either direction, and phase variations of ±30°.

```
DGM = getDGM(2,30,'tight');
F = umargin('F')
```

```
F =
  Uncertain gain/phase "F" with relative gain change in [0.5,2] and phase change of ±36.9 degrees
```

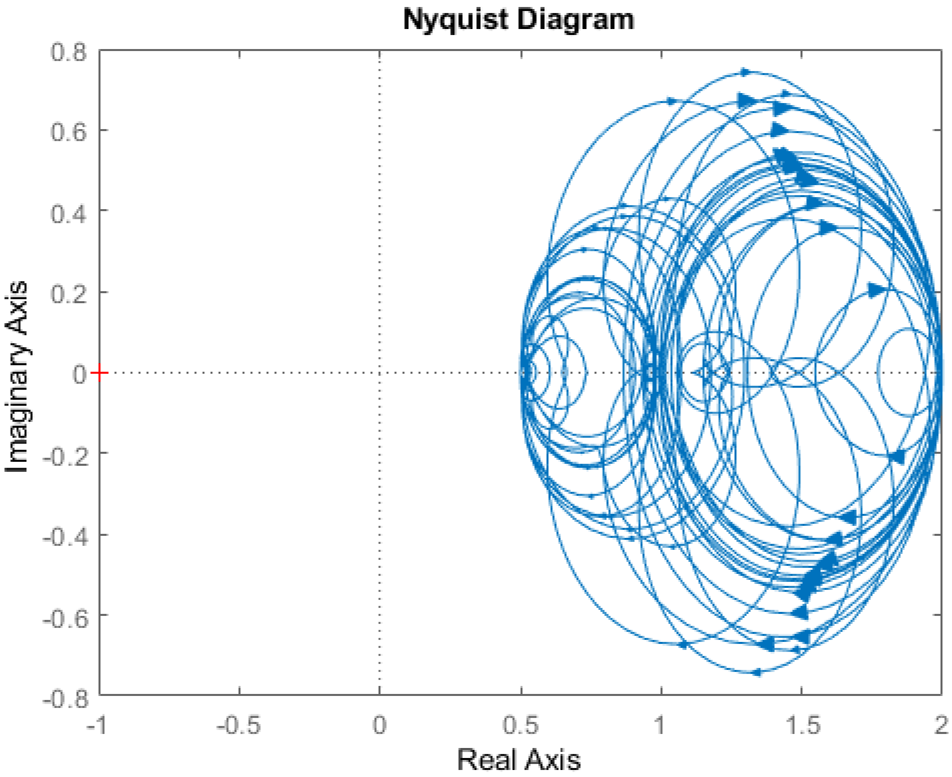
The samples of a umargin block are also state-space models.

```
Fs = usample(F,30);
size(Fs)
```

```
30x1 array of state-space models.
Each model has 1 outputs, 1 inputs, and 3 states.
```

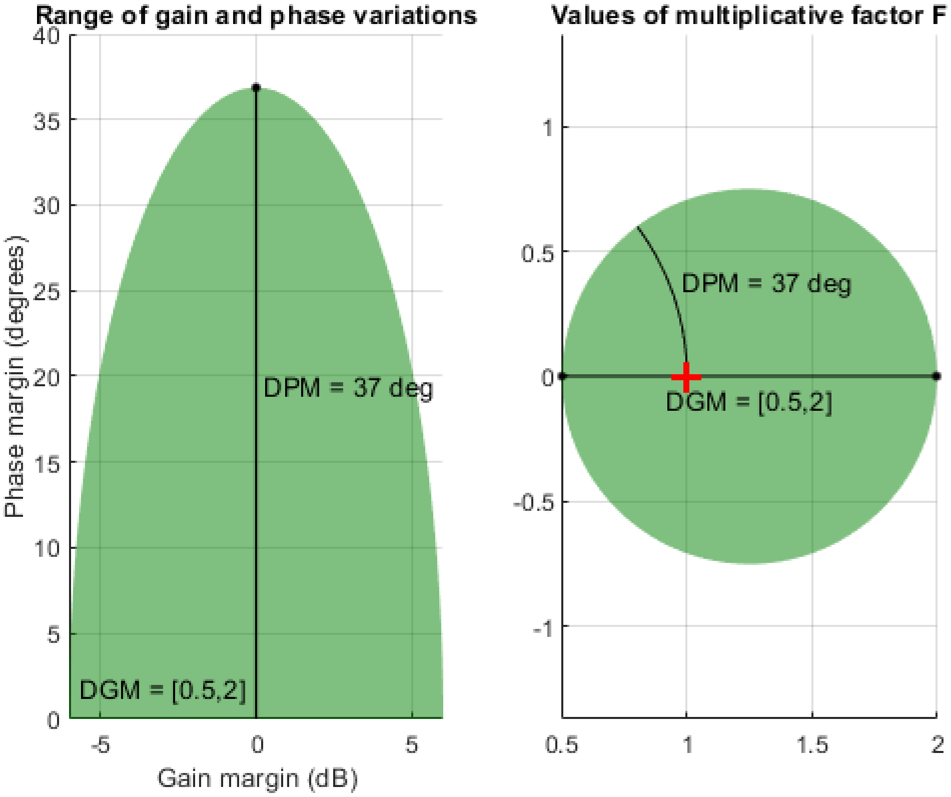
Plot the samples on the Nyquist plane.

```
nyquist(Fs)
```



The Nyquist plot of any sample of F stays within the disk of uncertainty modeled by F. To confirm this bound, use plot to examine the uncertainty disk. Compare the Nyquist plot above with the right side of the following plot.

```
plot(F)
```



For further details about the gain and phase uncertainty model, see `umargin`.

See Also

`usample` | `usubs`

Related Examples

- "Evaluate Uncertain Elements by Substitution" on page 1-25

Substitution by usubs

If an uncertain matrix or model object (`umat`, `uss`, `ufrd`) has many uncertain parameters, it is often useful to freeze some, but not all, of the uncertain parameters to specific values for analysis. The `usubs` command accomplishes this, and also allows more complicated substitutions for an element.

`usubs` accepts a list of element names and respective values to substitute for them. For example, can create three uncertain real parameters and use them to create a 2-by-2 uncertain matrix, `A`.

```
delta = ureal('delta',2);
eta = ureal('eta',6);
rho = ureal('rho',-1);
A = [3+delta+eta delta/eta;7+rho rho+delta*eta]
```

```
A =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    delta: Uncertain real, nominal = 2, variability = [-1,1], 2 occurrences
    eta: Uncertain real, nominal = 6, variability = [-1,1], 3 occurrences
    rho: Uncertain real, nominal = -1, variability = [-1,1], 1 occurrences

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

Use `usubs` to substitute the uncertain element named `delta` in `A` with the value 2.3, leaving all other uncertain elements intact. That the result, `B`, is an uncertain matrix with dependence only on `eta` and `rho`.

```
B = usubs(A,'delta',2.3)
```

```
B =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    eta: Uncertain real, nominal = 6, variability = [-1,1], 3 occurrences
    rho: Uncertain real, nominal = -1, variability = [-1,1], 1 occurrences

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

To set multiple elements, list individually, or group the values in a data structure. For instance, the following code creates identical uncertain matrices `B1` and `B2`. In each case, you replace `delta` by 2.3, and `eta` by the uncertain real parameter `A.Uncertainty.rho`.

```
B1 = usubs(A,'delta',2.3,'eta',A.Uncertainty.rho)
```

```
B1 =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    rho: Uncertain real, nominal = -1, variability = [-1,1], 4 occurrences

Type "B1.NominalValue" to see the nominal value, "get(B1)" to see all properties, and "B1.Uncerta
```

```
S.delta = 2.3;
S.eta = A.Uncertainty.rho;
B2 = usubs(A,S)
```

```
B2 =
  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    rho: Uncertain real, nominal = -1, variability = [-1,1], 4 occurrences

Type "B2.NominalValue" to see the nominal value, "get(B2)" to see all properties, and "B2.Uncerta
```

usubs ignores substitutions that do not match uncertain parameters in the model or matrix. For example, the following returns an uncertain matrix that is the same as A.

```
B3 = usubs(A,'fred',5);
```

Specifying the Substitution with Structures

An alternative syntax for usubs is to specify the substituted values in a structure, whose field names are the names of the elements being substituted with values. For example, create a structure NV with fields delta and eta. Set the values of these fields to be the desired values for substitution. Then perform the substitution with usubs.

```
NV.delta = 2.3;
NV.eta = A.Uncertainty.rho;
B4 = usubs(A,NV)
```

```
B4 =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    rho: Uncertain real, nominal = -1, variability = [-1,1], 4 occurrences

Type "B4.NominalValue" to see the nominal value, "get(B4)" to see all properties, and "B4.Uncerta
```

Here, B4 is the same as B1 and B2 above. Again, any superfluous fields are ignored. Therefore, adding an additional field gamma to NV does not alter the result of substitution.

```
NV.gamma = 0;
B5 = usubs(A,NV)
```

```
B5 =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    rho: Uncertain real, nominal = -1, variability = [-1,1], 4 occurrences

Type "B5.NominalValue" to see the nominal value, "get(B5)" to see all properties, and "B5.Uncerta
```

B5 is the same as B4.

Analysis commands such as wcgain, robstab, and usample all return substitutable values in this structure format.

Nominal and Random Values

To fix specified elements to their nominal values, use the replacement value 'Nominal'. To set an element to a random value, use 'Random'. For example, create a numeric matrix by fixing uncertain parameters in A: Set eta to its nominal value, set delta to a random value, and set rho to 6.5.

```
B6 = usubs(A,'eta','Nominal','delta','Random','rho',6.5)
```

```
B6 = 2×2
```

```
   11.6294    0.4382
   13.5000   22.2767
```

In the structure format, to set an uncertain element to its nominal value, set the corresponding value in the structure.

```
S = struct('eta',A.Uncertainty.eta.NominalValue,'rho',6.5);
B7 = usubs(A,S)
```

```
B7 = 

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    delta: Uncertain real, nominal = 2, variability = [-1,1], 2 occurrences

Type "B7.NominalValue" to see the nominal value, "get(B7)" to see all properties, and "B7.Uncerta
```

Use `usample` to set the remaining element to a random value.

```
B8 = usample(B7,'delta',1)
```

```
B8 = 2×2

   11.8116    0.4686
   13.5000   23.3695
```

See Also

`usample` | `usubs`

Related Examples

- “Sample Uncertain Systems” on page 1-39
- “Evaluate Uncertain Elements by Substitution” on page 1-25

Array Management for Uncertain Objects

All of the uncertain system classes (uss, ufrd) may be multidimensional arrays. This is intended to provide the same functionality as the LTI-arrays of the Control System Toolbox software. The command size returns a row vector with the sizes of all dimensions.

The first two dimensions correspond to the outputs and inputs of the system. Any dimensions beyond are referred to as the *array dimensions*. Hence, if szM = size(M), then szM(3:end) are sizes of the array dimensions of M.

For these types of objects, it is clear that the first two dimensions (system output and input) are interpreted differently from the 3rd, 4th, 5th and higher dimensions (which often model parametrized variability in the system input/output behavior).

umat objects are treated in the same manner. The first two dimensions are the rows and columns of the uncertain matrix. Any dimensions beyond are the *array dimensions*.

Reference Into Arrays

Suppose M is a umat, uss or ufrd, and that Yidx and Uidx are vectors of integers. Then

```
M(Yidx,Uidx)
```

selects the outputs (rows) referred to by Yidx and the inputs (columns) referred to by Uidx, preserving all of the array dimensions. For example, if size(M) equals [4 5 3 6 7], then (for example) the size of M([4 2],[1 2 4]) is [2 3 3 6 7].

If size(M,1)==1 or size(M,2)==1, then single indexing on the inputs or outputs (rows or columns) is allowed. If Sidx is a vector of integers, then M(Sidx) selects the corresponding elements. All array dimensions are preserved.

If there are K array dimensions, and idx1, idx2, ..., idxK are vectors of integers, then

```
G = M(Yidx,Uidx,idx1,idx2,...,idxK)
```

selects the outputs and inputs referred to by Yidx and Uidx, respectively, and selects from each array dimension the “slices” referred to by the idx1, idx2,..., idxK index vectors. Consequently, size(G,1) equals length(Yidx), size(G,2) equals length(Uidx), size(G,3) equals length(idx1), size(G,4) equals length(idx2), and size(G,K+2) equals length(idxK).

If M has K array dimensions, and less than K index vectors are used in doing the array referencing, then the MATLAB convention for single indexing is followed. For instance, suppose size(M) equals [3 4 6 5 7 4]. The expression

```
G = M([1 3],[1 4],[2 3 4],[5 3 1],[8 10 12 2 4 20 18])
```

is valid. The result has size(G) equals [2 2 3 3 7] . The last index vector [8 10 12 2 4 20 18] is used to reference into the 7-by-4 array, preserving the order dictated by MATLAB single indexing (e.g., the 10th element of a 7-by-4 array is the element in the (3,2) position in the array).

Note that if M has either one output (row) or one input (column), *and* M has array dimensions, then it is not allowable to combine single indexing in the output/input dimensions along with indexing in the array dimensions. This will result in an ambiguity in how to interpret the second index vector in the expression (i.e., “does it correspond to the input/output reference, or does it correspond to the first array dimension?”).

See Also

Related Examples

- “Model Arrays” (Control System Toolbox)

Create Arrays with stack and cat Functions

An easy manner to create an array is with `stack`. Create a [4-by-1] `umat` array by stacking four 1-by-3 `umat` objects with the `stack` command. The first argument of `stack` specifies in which array dimension the stacking occurs. In the example below, the stacking is done is the 1st array dimension, hence the result is a 1-by-3-by-4-by-1 `umat`, referred to as a 4-by-1 `umat` array.

```
a = ureal('a',4);
b = ureal('b',2);
M = stack(1,[a b 1],[-a -b 4+a],[4 5 6],[a 0 0])
UMAT: 1 Rows, 3 Columns [array, 4 x 1]
  a: real, nominal = 4, variability = [-1  1], 1 occurrence
  b: real, nominal = 2, variability = [-1  1], 1 occurrence
size(M)
ans =
     1     3     4
arraysize(M)
ans =
     4     1
```

Check that result is valid. Use referencing to access parts of the [4-by-1] `umat` array and compare to the expected values. The first 4 examples should all be arrays full of 0 (zeros). The last two should be the value 5, and the uncertain real parameter `a`, respectively.

```
simplify(M(:,:,1) - [a b 1])
ans =
     0     0     0
simplify(M(:,:,2) - [-a -b 4+a])
ans =
     0     0     0
simplify(M(:,:,3) - [4 5 6])
ans =
     0     0     0
simplify(M(:,:,4) - [a 0 0])
ans =
     0     0     0
simplify(M(1,2,3))  % should be 5
ans =
     5
simplify(M(1,3,2)-4)
Uncertain Real Parameter: Name a, NominalValue 4, variability = [-1  1]
```

You can create a random 1-by-3-by-4 double matrix and stack this with `M` along the second array dimension, creating a 1-by-3-by-4-by-2 `umat`.

```
N = randn(1,3,4);
M2 = stack(2,M,N);
size(M2)
ans =
     1     3     4     2
arraysize(M2)
ans =
     4     2
```

As expected, both `M` and `N` can be recovered from `M2`.

```
d1 = simplify(M2(:,:,:,1)-M);
d2 = simplify(M2(:,:,:,2)-N);
```

```
[max(abs(d1(:))) max(abs(d2(:)))]
ans =
     0     0
```

It is also possible to stack `M` and `N` along the 1st array dimension, creating a 1-by-3-by-8-by-1 `umat`.

```
M3 = stack(1,M,N);
size(M3)
ans =
     1     3     8
arraysize(M3)
ans =
     8     1
```

As expected, both `M` and `N` can be recovered from `M3`.

```
d3 = simplify(M3(:,:,1:4)-M);
d4 = simplify(M3(:,:,5:8)-N);
[max(abs(d3(:))) max(abs(d4(:)))]
ans =
     0     0
```

Create Arrays by Assignment

Arrays can be created by direct assignment. As with other MATLAB classes, there is no need to preallocate the variable first. Simply assign elements – all resizing is performed automatically.

For instance, an equivalent construction to

```
a = ureal('a',4);
b = ureal('b',2);
M = stack(1,[a b 1],[-a -b 4+a],[4 5 6],[a 0 0]);
is
Mequiv(1,1,1) = a;
Mequiv(1,2,1) = b;
Mequiv(1,3,1) = 1;
Mequiv(1,:,4) = [a 0 0];
Mequiv(1,:,2:3) = stack(1,[-a -b 4+a],[4 5 6]);
```

The easiest manner for you to verify that the results are the same is to subtract and simplify,

```
d5 = simplify(M-Mequiv);
max(abs(d5(:)))
ans =
     0
```

Binary Operations with Arrays

Most operations simply cycle through the array dimensions, doing pointwise operations. Assume A and B are umat (or uss, or ufrd) arrays with identical array dimensions (slot 3 and beyond). The operation C = fcn(A,B) is equivalent to looping on k1, k2, ..., setting

```
C(:,:,k1,k2,...) = fcn(A(:,:,k1,k2,...),B(:,:,k1,k2,...))
```

The result C has the same array dimensions as A and B. The user is required to manage the extra dimensions (i.e., keep track of what they mean). Methods such as permute, squeeze and reshape are included to facilitate this management.

In general, any binary operation requires that the extra-dimensions are compatible. The umat, uss and ufrd objects allow for slightly more flexible interpretation of this. For illustrative purposes, consider a binary operation involving variables *A* and *B*. Suppose the array dimensions of *A* are $n_1 \times \ldots \times n_{l_A}$ and that the array dimensions of *B* are $m_1 \times \ldots \times m_{l_B}$. By MATLAB convention, the infinite number of singleton (i.e., 1) trailing dimensions are not listed. The compatibility of the extra dimensions is determined by the following rule: If *lA* = *lB*, then pad the shorter dimension list with trailing 1's. Now compare the extra dimensions: In the *k-th* dimension, it must be that one of 3 conditions hold: *nk* = *mk*, or *nk* = 1, or *mk* = 1. In other words, non-singleton dimensions must exactly match (so that the pointwise operation can be executed), and singleton dimensions match with anything, implicitly through a repmat.

Sample Uncertain Elements to Create Arrays

A common way to generate an array is to sample the uncertain elements of an uncertain object. This example shows how to generate arrays by taking random samples of a `umat` uncertain matrix that has two uncertain elements. (To generate arrays by sampling at specific values, use `usubs`.)

Create an uncertain matrix.

```
a = ureal('a',4);
b = ureal('b',2);
M = [a b;b*b a/b;1-b 1+a*b]
```

```
M =

  Uncertain matrix with 3 rows and 2 columns.
  The uncertainty consists of the following blocks:
    a: Uncertain real, nominal = 4, variability = [-1,1], 3 occurrences
    b: Uncertain real, nominal = 2, variability = [-1,1], 6 occurrences

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

Sample the uncertain real parameter `b` in the matrix `M`, at 20 random points within its range.

```
[Ms,bvalues] = usample(M,'b',20);
```

This results in an array of 20 3-by-2 `umat` matrices, with only one uncertain element, `a`. The uncertain element `b` of `M` has been sampled out, leaving a new array dimension in its place.

```
Ms
```

```
Ms =

  20x1 array of uncertain matrices with 3 rows, 2 columns, and the following uncertain blocks:
    a: Uncertain real, nominal = 4, variability = [-1,1], 3 occurrences

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

Additionally, `bvalues` is a structure containing the corresponding sampled values of `b`.

```
bvalues
```

```
bvalues=20×1 struct array with fields:
    b
```

Next, sample the remaining uncertain real parameter `a` in the matrix `Ms`. This removes the second uncertain block, resulting in a 3-by-2-by-20-by-15 `double`.

```
[Mss,avalues] = usample(Ms,'a',15);
size(Mss)
```

```
ans = 1×4

     3     2    20    15
```

You can also sample multiple parameters at once. The following operation returns `Mss1`, which is identical to `Mss`.

```
[Mss1,values] = usample(M,'b',20,'a',15);
```

Rather than sampling each variable (a and b) independently, generating a 20-by-15 grid in a 2-dimensional space, you can sample the two-dimensional space directly. Sample the 2-dimensional space with 800 points.

```
[Ms2d,values] = usample(M,{'a' 'b'},800);
```

Ms2d is a 3-by-2-by-800 umat array, where each entry corresponds to a different randomly selected (a,b) pair. The structure array values contains these (a,b) values.

```
values
```

```
values=800×1 struct array with fields:
    a
    b
```

See Also

umat | usample | usubs

Related Examples

- “Substitute Uncertain Elements to Create Arrays” on page 1-66

Substitute Uncertain Elements to Create Arrays

You can generate an array from an uncertain object by replacing the uncertain elements with specified values. There are several ways to do this using `usubs`.

Create a 3-by-2 uncertain matrix using two uncertain real parameters.

```
a = ureal('a',4);
b = ureal('b',2);
M = [a b;b*b a/b;1-b 1+a*b];
```

Evaluate the matrix at four different combinations of values for the uncertain parameters `a` and `b`.

```
avals = [1;2;3;4];
bvals = [10;11;12;13];
M1 = usubs(M,'a',avals,'b',bvals);
```

This command evaluates `M` for the four different `(a,b)` combinations `(1,10)`, `(2,11)`, and so on. Therefore, `M1` is a 3-by-2-by-4 double array containing the four evaluated values of M along its last dimension.

```
size(M1)
```

```
ans = 1×3

     3     2     4
```

For more examples, see `usubs`.

See Also

`umat` | `usample` | `usubs`

Related Examples

- “Sample Uncertain Elements to Create Arrays” on page 1-64

Create Arrays with gridureal

The command `gridureal` enables uniform sampling of specified uncertain real parameters within an uncertain object. It is a specialized case of `usubs`.

`gridureal` removes a specified uncertain real parameter and adds an array dimension (to the end of the existing array dimensions). The new array dimension represents the uniform samples of the uncertain object in the specified uncertain real parameter range.

Create a 2-by-2 uncertain matrix with three uncertain real parameters.

```
a = ureal('a',3,'Range',[2.5 4]);
b = ureal('b',4,'Percentage',15);
c = ureal('c',-2,'Plusminus',[-1 .3]);
M = [a b;b c]
UMAT: 2 Rows, 2 Columns
  a: real, nominal = 3, range = [2.5  4], 1 occurrence
  b: real, nominal = 4, variability = [-15  15]%, 2 occurrences
  c: real, nominal = -2, variability = [-1  0.3], 1 occurrence
```

Grid the uncertain real parameter `b` in `M` with 100 points. The result is a `umat` array, with dependence on uncertain real parameters `a` and `c`.

```
Mgrid1 = gridureal(M,'b',100)
UMAT: 2 Rows, 2 Columns [array, 100 x 1]
  a: real, nominal = 3, range = [2.5  4], 1 occurrence
  c: real, nominal = -2, variability = [-1  0.3], 1 occurrence
```

Operating on the uncertain matrix `M`, grid the uncertain real parameter `a` with 20 points, the uncertain real parameter `b` with 12 points, and the uncertain real parameter `c` with 7 points, The result is a 2-by-2-by20-by-12-by7 `double` array.

```
Mgrid3 = gridureal(M,'a',20,'b',12,'c',7);
size(Mgrid3)
ans =
     2     2    20    12     7
```

Create Arrays with repmat

The MATLAB command repmat is used to replicate and tile arrays. It works on the built-in objects of MATLAB, namely double, char, as well as the generalized container objects cell and struct. The identical functionality is provided for replicating and tiling uncertain elements (ureal, ultidyn, etc.) and umat objects.

You can create an uncertain real parameter, and replicate it in a 2-by-3 uncertain matrix. Compare to generating the same uncertain matrix through multiplication.

```
a = ureal('a',5);
Amat = repmat(a,[2 3])
UMAT: 2 Rows, 3 Columns
  a: real, nominal = 5, variability = [-1  1], 1 occurrence
Amat2 = a*ones(2,3);
simplify(Amat-Amat2)
ans =
     0     0     0
     0     0     0
```

Create a [4-by-1] umat array by stacking four 1-by-3 umat objects with the stack command. Use repmat to tile this 1-by-3-by-4-by-1 umat, into a 2-by-3-by-8-by-5 umat.

```
a = ureal('a',4);
b = ureal('b',2);
M = stack(1,[a b 1],[-a -b 4+a],[4 5 6],[a 0 0]);
size(M)
ans =
     1     3     4
Mtiled = repmat(M,[2 1 2 5])
UMAT: 2 Rows, 3 Columns [array, 8 x 5]
  a: real, nominal = 4, variability = [-1  1], 1 occurrence
  b: real, nominal = 2, variability = [-1  1], 1 occurrence
Verify the equality of M and a few certain tiles of Mtiled.
d1 = simplify(M-Mtiled(2,:,5:8,3));
d2 = simplify(M-Mtiled(1,:,1:4,2));
d3 = simplify(M-Mtiled(2,:,1:4,5));
[max(abs(d1(:))) max(abs(d2(:))) max(abs(d3(:)))]
ans =
     0     0     0
```

Note that repmat never increases the complexity of the representation of an uncertain object. The number of occurrences of each uncertain element remains the same, regardless of the extent of the replication and tiling.

Create Arrays with repsys

Replicate and tile uncertain state-space (uss) and uncertain frequency response data (ufrd) models with repsys. The syntax and behavior are the same as the manner in which repmat is used to replicate and tile matrices. The syntax and behavior of repsys for uss and ufrd objects are the same as the traditional repsys which operates on ss and frd objects. Just as in those cases, the uncertain version of repsys also allows for diagonal tiling.

Using permute and ipermute

The commands permute and ipermute are generalizations of transpose, which exchanges the rows and columns of a two-dimensional matrix.

permute(A,ORDER) rearranges the dimensions of A so that they are in the order specified by the vector ORDER. The array produced has the same values of A but the order of the subscripts needed to access any particular element are rearranged as specified by ORDER. The elements of ORDER must be a rearrangement of the numbers from 1 to N.

All of the uncertain objects are essentially 2-dimensional (output and input) operators with array dependence. This means that the first 2 dimensions are treated differently from dimensions 3 and beyond. It is not permissible to permute across these groups.

For uss and ufrd, the restriction is built into the syntax. The elements of the ORDER vector only refer to array dimensions. Therefore, there is no possibility of permute across these dimensions. In you need to permute the first two dimensions, use the command transpose instead.

For umat, the restriction is enforced in the software. The elements of the ORDER vector refer to all dimensions. However, the first two elements of ORDER must be a rearrangement of the numbers 1 and 2. The remaining elements of ORDER must be a rearrangement of the numbers 3 through N. If either of those conditions fail, an error is generated. Hence, for umat arrays, either permute or transpose can be used to effect the transpose operation.

Decomposing Uncertain Objects

Each uncertain model or matrix (such as `uss`, `genss`, `ufrd`, or `umat`,) is a generalized feedback connection (`lft`) of a not-uncertain object (e.g., `double`, `ss`, `frd`) with a diagonal augmentation of uncertain elements (`ureal`, `ultidyn`, `umargin`, `ucomplex`, `ucomplexm`, `udyn`). In robust control jargon, if the uncertain elements are normalized, this decomposition is often called "the M/D form."

The purpose of the uncertain objects (`ureal`, `ultidyn`, `umat`, `uss`, etc.) is to hide this underlying decomposition, and allow the user to focus on modeling and analyzing uncertain systems, rather than the details of correctly propagating the M/D representation in manipulations. Nevertheless, advanced users may want access to the familiar M/D form. The command `lftdata` accomplishes this decomposition.

Since `ureal`, `umargin`, `ucomplex` and `ucomplexm` do not have their `NominalValue` necessarily at zero, and in the case of `ureal` and `umargin` objects, are not necessarily symmetric about the `NominalValue`, some details are required in describing the decomposition.

Normalizing Functions for Uncertain Elements

Associated with each uncertain element is a normalizing function. The normalizing function maps the uncertain element into a normalized uncertain element. Regardless of element type, as the uncertain element varies over its range, the absolute value of the normalizing function (or norm, in the matrix case) varies from 0 and 1.

Uncertain Real Parameter (ureal)

If ρ is an uncertain real parameter, with range `[L R]` and nominal value `N`, then the normalizing function F is

$$F(\rho) = \frac{A + B\rho}{C + D\rho}$$

with the property that for all ρ satisfying $L \le \rho \le R$, it follows that $-1 \le F(\rho) \le 1$, moreover, $F(L) = -1$, $F(N) = 0$, and $F(R) = 1$. If the nominal value is centered in the range, then it is easy to conclude that

$$A = \frac{R + L}{R - L}$$

$$B = \frac{2}{R - L}$$

$$C = 1$$

$$D = 0.$$

It is left as an algebra exercise for the user to work out the various values for `A, B, C` and `D` when the nominal value is not centered.

Uncertain Linear Time-Invariant Dynamic Uncertainty (ultidyn)

If E is an uncertain gain-bounded, linear, time-invariant dynamic uncertainty, with gain-bound β, then the normalizing function F is

$$F(E) = \frac{1}{\beta}E.$$

If E is an uncertain positive-real, linear, time-invariant dynamic uncertainty, with positivity bound β, then the normalizing function F is

$$F(E) = \left[I - \alpha\left(E - \frac{\beta}{2}I\right)\right]\left[I + \alpha\left(E - \frac{\beta}{2}I\right)\right]^{-1}$$

where $\alpha = 2|\beta| + 1$.

Uncertain Gain and Phase (umargin)

For a `umargin` block $Q(s)$, the normalized value $F(Q(s)) = \delta(s)$, where $\delta(s)$ is a gain-bounded dynamic uncertainty, normalized so that it always varies within the unit disk ($||\delta||_\infty < 1$). In other words, $F(Q)$ is a unit-gain `ultidyn` uncertain element.

The actual values of Q map to the unit-gain δ via the parameterization

$$Q(s) = \frac{1 + \alpha[(1 - E)/2]\delta(s)}{1 - \alpha[(1 + E)/2]\delta(s)}.$$

For details about this parameterization, see `umargin`.

Uncertain Complex Parameters (ucomplex)

The normalizing function for an uncertain complex parameter ξ, with nominal value C and radius γ, is

$$F(\xi) = \frac{1}{\gamma}(\xi - C).$$

Uncertain Complex Matrices (umat)

The normalizing function for uncertain complex matrices H, with nominal value N and weights W_L and W_R is

$$F(H) = W_L^{-1}(H - N)W_R^{-1}$$

Properties of the Decomposition

Take an uncertain object A, dependent on:

- Uncertain real parameters $\rho_1,\ldots,\rho_{N_\rho}$
- Uncertain complex parameters $\xi_1,\ldots,\xi_{N_\xi}$
- Uncertain complex matrices $H_1,\ldots,H_{N_H}$
- Uncertain gain-bounded linear, time-invariant dynamics $E_1,\ldots,E_{N_E}$
- Uncertain positive-real linear, time-invariant dynamics $P_1,\ldots,P_{N_P}$
- Uncertain gain and phase $Q_1,\ldots,Q_{N_Q}$

Write $A(\rho,\xi,H,E,P,Q)$ to indicate this dependence. Using `lftdata`, A can be decomposed into two separate pieces: M and $\Delta(\rho,\xi,H,E,P,Q)$ with the following properties:

- M is certain (i.e., if A is `uss`, then M is `ss`; if A is `umat`, then M is `double`; if A is `ufrd`, then M is `frd`).
- Δ is always a `umat`, depending on the same uncertain elements as A, with ranges, bounds, weights, etc., unaltered.

- The form of Δ is block diagonal, with elements made up of the normalizing functions acting on the individual uncertain elements:

$$\Delta(\rho,\xi,H,E,P,Q) = \begin{bmatrix} F(\rho) & 0 & 0 & 0 & 0 & 0 \\ 0 & F(\xi) & 0 & 0 & 0 & 0 \\ 0 & 0 & F(H) & 0 & 0 & 0 \\ 0 & 0 & 0 & F(E) & 0 & 0 \\ 0 & 0 & 0 & 0 & F(P) & \\ 0 & 0 & 0 & 0 & 0 & F(Q) \end{bmatrix}.$$

- $A(\rho,\xi,H,E,P,Q)$ is given by a linear fractional transformation of M and $\Delta(\rho,\xi,H,E,P,Q)$,

$$A(\rho,\xi,H,E,P,Q) = M_{22} + M_{21}\Delta(\rho,\xi,H,E,P,Q)[I - M_{11}\Delta(\rho,\xi,H,E,P,Q)]^{-1}M_{12}\,.$$

The order of the normalized elements making up `A` is not the simple order shown above. It is actually the same order as given by the command `fieldnames(M.Uncertainty)`, as shown in the following example.

Decompose Uncertain Model Using lftdata

You decompose an uncertain model into a fixed certain part and normalized uncertain part using the `lftdata` command. To see how this command works, create a 2-by-2 uncertain matrix (`umat`) using three uncertain real parameters.

```
delta = ureal('delta',2);
eta = ureal('eta',6);
rho = ureal('rho',-1);
A = [3+delta+eta delta/eta;7+rho rho+delta*eta]
```

```
A =

  Uncertain matrix with 2 rows and 2 columns.
  The uncertainty consists of the following blocks:
    delta: Uncertain real, nominal = 2, variability = [-1,1], 2 occurrences
    eta: Uncertain real, nominal = 6, variability = [-1,1], 3 occurrences
    rho: Uncertain real, nominal = -1, variability = [-1,1], 1 occurrences

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

The `umat` `A` depends on two occurrences of `delta`, three occurrences of `eta`, and one occurrence of `rho`.

Decompose `A` into `M` and `Delta`.

```
[M,Delta] = lftdata(A);
```

`M` is a numeric matrix.

```
M
```

```
M = 8×8

         0         0         0   -0.1667         0         0    1.0000    0.1667
         0         0         0         0    1.0000         0         0    6.0000
         0         0         0         0         0         0    1.0000         0
         0         0         0   -0.1667         0         0         0    0.1667
         0         0         0         0         0         0         0    1.0000
         0         0         0         0         0         0    1.0000    1.0000
    1.0000         0    1.0000   -0.3333         0         0   11.0000    0.3333
         0    1.0000         0         0    2.0000    1.0000    6.0000   11.0000
```

`Delta` is a `umat` with the same uncertainty dependence as `A`.

```
Delta
```

```
Delta =

  Uncertain matrix with 6 rows and 6 columns.
  The uncertainty consists of the following blocks:
    delta: Uncertain real, nominal = 2, variability = [-1,1], 2 occurrences
    eta: Uncertain real, nominal = 6, variability = [-1,1], 3 occurrences
    rho: Uncertain real, nominal = -1, variability = [-1,1], 1 occurrences

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

To examine some of the characteristics of `Delta`, sample it at three points. Note that:

- The sampled value of `Delta` is always diagonal.
- The sampled values always range between -1 and 1, because `Delta` is normalized.
- The sampled matrices each contain three independent values. Duplication of the entries is consistent with the dependence of `Delta` and `A` on the three uncertain real parameters.

```
usample(Delta,3)
```

```
ans = 
ans(:,:,1) =

    0.6294         0         0         0         0         0
         0    0.6294         0         0         0         0
         0         0    0.8268         0         0         0
         0         0         0    0.8268         0         0
         0         0         0         0    0.8268         0
         0         0         0         0         0   -0.4430


ans(:,:,2) =

    0.8116         0         0         0         0         0
         0    0.8116         0         0         0         0
         0         0    0.2647         0         0         0
         0         0         0    0.2647         0         0
         0         0         0         0    0.2647         0
         0         0         0         0         0    0.0938


ans(:,:,3) =

   -0.7460         0         0         0         0         0
         0   -0.7460         0         0         0         0
         0         0   -0.8049         0         0         0
         0         0         0   -0.8049         0         0
         0         0         0         0   -0.8049         0
         0         0         0         0         0    0.9150
```

Verify that the maximum gain of `Delta` is 1.

```
maxnorm = wcnorm(Delta)
```

```
maxnorm = struct with fields:
    LowerBound: 0
    UpperBound: 1.0008
```

Finally, verify that `lft(Delta,M)` is the same as `A`. To do so, take the difference, and use the `'full'` option in `simplify` to remove redundant dependencies on uncertain elements.

```
simplify(lft(Delta,M)-A,'full')
```

```
ans = 2×2

     0     0
     0     0
```

Advanced Syntax of lftdata

Even for the advanced user, the variable `Delta` will actually not be that useful, as it is still a complex object. On the other hand, its internal structure is described completely using a 3rd (and 4th) output argument.

```
[M,Delta,BlkStruct,NormUnc] = lftdata(A);
```

The rows of `BlkStruct` correspond to the uncertain elements named in `fieldnames(A.Uncertainty)`. The elements of `BlkStruct` describe the size, type and number-of-copies of the uncertain elements in `A`, and implicitly delineate the exact block-diagonal structure of `Delta`. Note that the range/bound information about each uncertain element is not included in BlkStruct.

```
BlkStruct(1)
```

```
ans = struct with fields:
           Name: 'delta'
           Size: [1 1]
           Type: 'ureal'
    Occurrences: 2
       Simplify: 2
```

```
BlkStruct(2)
```

```
ans = struct with fields:
           Name: 'eta'
           Size: [1 1]
           Type: 'ureal'
    Occurrences: 3
       Simplify: 2
```

```
BlkStruct(3)
```

```
ans = struct with fields:
           Name: 'rho'
           Size: [1 1]
           Type: 'ureal'
    Occurrences: 1
       Simplify: 2
```

Together, these entries mean that `Delta` is a block diagonal augmentation of the normalized version of the three uncertain elements.

The first element is named `'delta'`. It is 1-by-1; it is of class `ureal`; and there are two copies diagonally augmented.

The second element is named `'eta'`. It is 1-by-1; it is of class `ureal`; and there are three copies diagonally augmented.

The third element is named `'rho'`. It is 1-by-1; it is of class `ureal`; and there is one copy,

The fourth output argument of `lftdata` contains a cell array of normalized uncertain elements. The cell array contains as many occurrences of each element as there are occurrences in the original uncertain object `A`.

```
size(NormUnc)
```

```
ans = 1×2

     6     1
```

```
NormUnc{1}
```

```
ans = 
  Uncertain real parameter "deltaNormalized" with nominal value 0 and variability [-1,1].
```

```
isequal(NormUnc{2},NormUnc{1})
```

```
ans = logical
   1
```

```
NormUnc{3}
```

```
ans = 
  Uncertain real parameter "etaNormalized" with nominal value 0 and variability [-1,1].
```

```
isequal(NormUnc{4},NormUnc{3})
```

```
ans = logical
   1
```

```
isequal(NormUnc{5},NormUnc{3})
```

```
ans = logical
   1
```

```
NormUnc{6}
```

```
ans = 
  Uncertain real parameter "rhoNormalized" with nominal value 0 and variability [-1,1].
```

Each normalized element has `'Normalized'` appended to its original name to avoid confusion. When normalized,

- `ureal` objects have nominal value of 0, and range from –1 to 1.
- `ultidyn` objects are norm bounded, with norm bound of 1.
- `umargin` objects are converted to norm-bounded `ultidyn` objects with norm bound of 1.
- `ucomplex` objects have nominal value of 0, and radius 1.
- `ucomplexm` objects have nominal value of 0, and identity matrices for each of the `WL` and `WR` weights.

The possible behaviors of `Delta` and `blkdiag(NormUnc{:})` are the same. Consequently, the possible behaviors of `A` and `lft(blkdiag(NormUnc{:}),M)` are the same.

Hence, by manipulating `M`, `BlkStruct` and `NormUnc`, a you can have direct access to all of the linear fractional transformation details, and can work at the level of the theorems and algorithms that underlie the methods.

See Also

`actual2normalized` | `lftdata` | `normalized2actual` | `uscale`

Related Examples

- “Introduction to Uncertain Elements” on page 1-2

Generalized Robustness Analysis

Stability Analysis Using Disk Margins

Disk margins quantify the stability of a closed-loop system against gain or phase variations in the open-loop response. In disk-based margin calculations, the software models such variations as disk-shaped multiplicative uncertainty on the open-loop transfer function. The disk margin measures how much uncertainty the loop can tolerate before going unstable.

That uncertainty amount corresponds to minimum gain and phase margins. The disk-based gain margin *DGM* is the amount by which the loop gain can increase or decrease without loss of stability, in absolute units. The disk-based phase margin *DPM* is the amount by which the loop phase can increase or decrease without loss of stability, in degrees. These disk-based margins take into account all frequencies and loop interactions. Therefore, disk-based margin analysis provides a stronger guarantee of stability than the classical gain and phase margins.

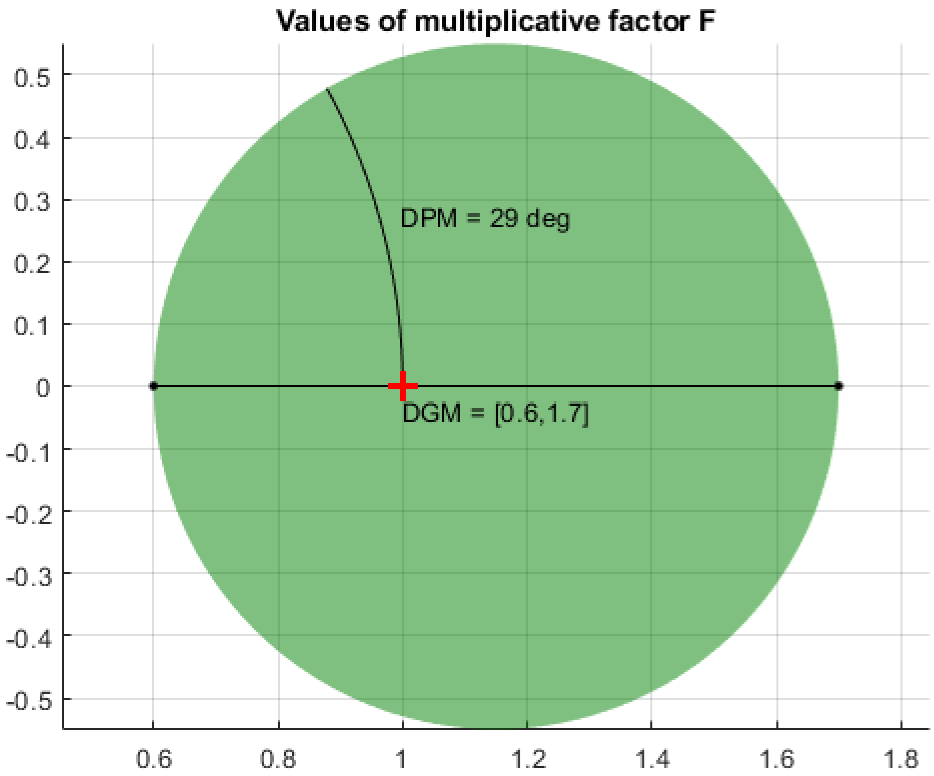
Robust Control Toolbox provides tools to:

- Analyze system stability against gain and phase variations. Use `diskmargin` to compute the disk-based gain and phase margins of SISO and MIMO feedback loops.
- Model gain and phase uncertainty. Use the `umargin` control design block to analyze the effect of gain and uncertainty on system performance and stability.

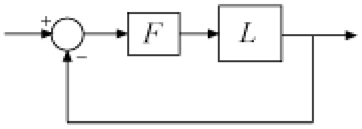
Modeling Gain and Phase Variations

Both `umargin` and `diskmargin` represent gain and phase variation as a multiplicative complex factor *F*(*s*), replacing the nominal open-loop response *L*(*s*) with *L*(*s*)**F*(*s*). The factor *F* takes values in a disk that includes the nominal value *F* = 1. This multiplicative factor models both gain and phase variations. For instance, the following plot shows one such disk in the complex plane.

```
DGM = [0.6,1.7];
diskmarginplot(DGM,'disk')
```



The values in this disk encompass relative gain-only variations in the range `DGM` = [0.6,1.7], or ±4 dB. They also represent absolute phase-only variations of `DPM` = [–29,29], or ±29°. Consider the following closed-loop system, with nominal loop transfer L and unit feedback.



If this feedback loop remains stable for all values of F in the disk shown in the previous plot, then the disk-based gain margin of L is at least `DGM`, and the disk-based phase margin is at least `DPM`.

Both umargin and diskmargin model gain and phase uncertainty with a family of disks described by two parameters, α and E. For SISO systems, the disk is parameterized by:

$$F = \frac{1 + \alpha[(1 - E)/2]\delta}{1 - \alpha[(1 + E)/2]\delta}.$$

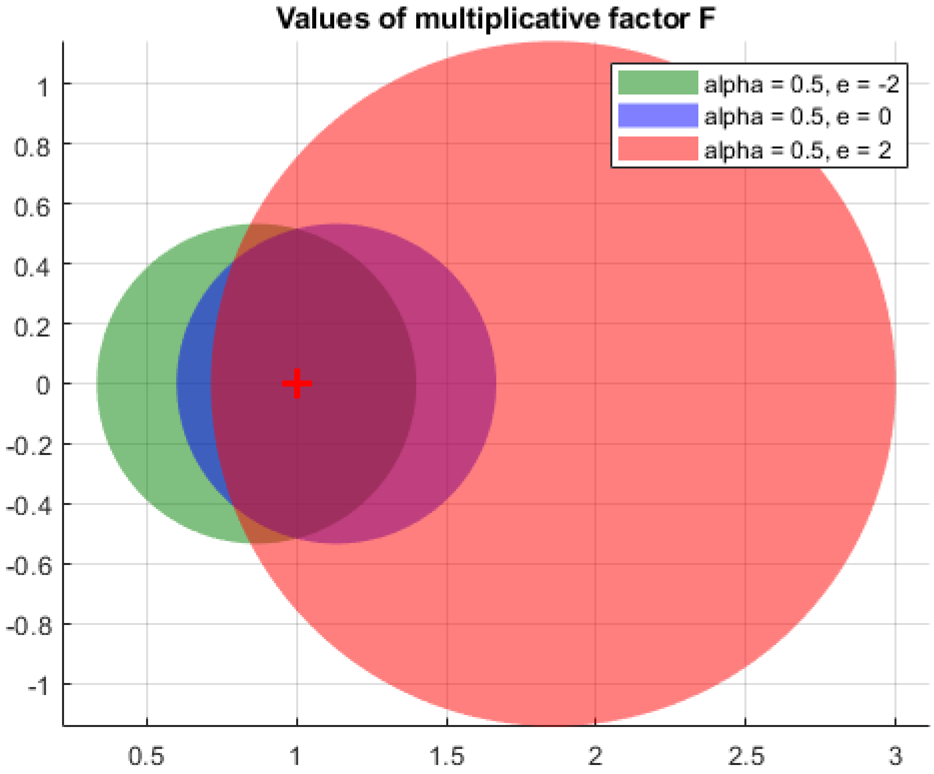
In this model,

- δ is the normalized uncertainty (an arbitrary complex value in the unit disk $|\delta| < 1$).
- α sets the amount of gain and phase variation modeled by F. For fixed E, the parameter α controls the size of the disk. For $\alpha = 0$, the multiplicative factor is 1, corresponding to the nominal L.

- *E*, called the *eccentricity*, skews the modeled uncertainty toward gain increase or gain decrease.

Each *α*,*E* pair corresponds to a disk that models a particular gain-variation range *DGM* = [*gmin*,*gmax*], given by the points where the disk intercepts the real (*x*) axis. The corresponding phase variation *DPM* is determined by the angle between the real axis and a line through the origin and tangent to the disk. Thus you can describe a modeled set of gain and phase variations entirely by either the two values *α*,*E* or the two values *DGM* = [*gmin*,*gmax*]. *E* = 0 models a balanced gain variation with [*gmin*,*gmax*] such that *gmin* = 1/*gmax*. When *E* < 0, then *F* represents a larger gain decrease than increase (*gmin* < 1/*gmax*). Conversely *E* > 0 represents a larger gain increase than decrease. For instance, consider the disks parameterized by *α* = 0.5 and three different eccentricities, *E* = –2, 0, and 2.

```
diskmarginplot(0.5,[-2 0 2],'disk')
```

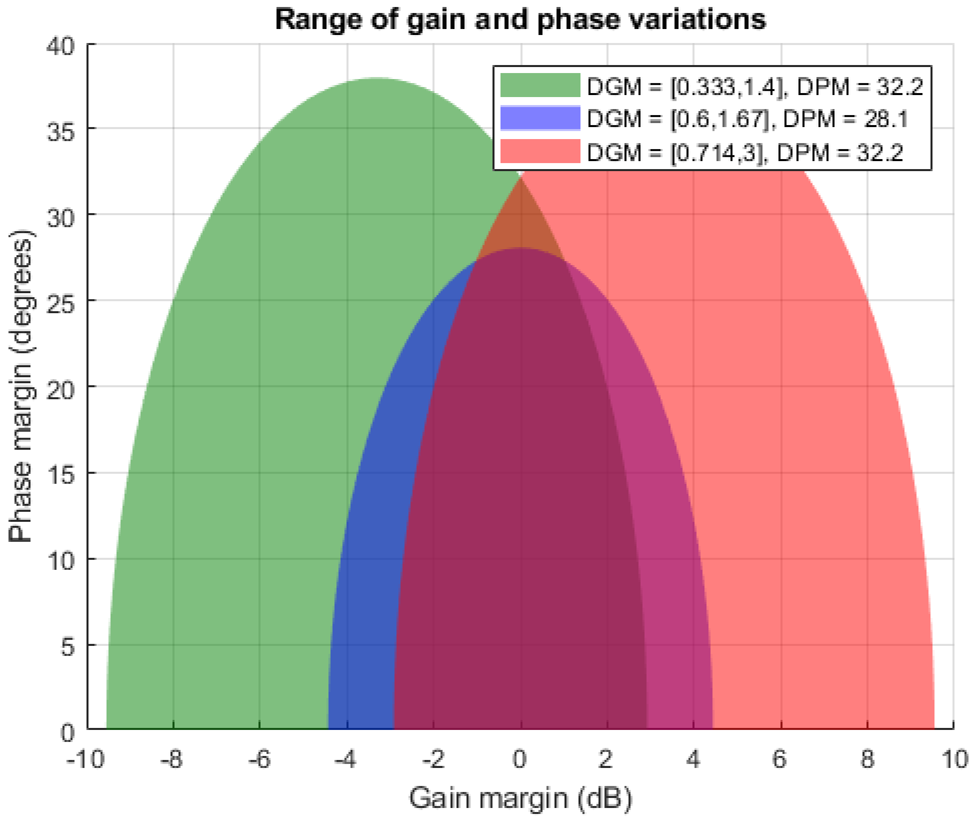


Each *α*,*E* pair corresponds to a disk that models a different gain-variation range *DGM* = [*gmin*,*gmax*]. Examine the gain variations that correspond to each of these three disks.

```
Ranges = dm2gm(0.5,[-2  0 2])
```

```
Ranges = 3×2

    0.3333    1.4000
    0.6000    1.6667
    0.7143    3.0000
```

```
diskmarginplot(Ranges)
```



The balanced $E = 0$ range is symmetric around the nominal value, allowing the gain to increase or decrease by a factor of about 1.67. The negative E value corrsponds to more gain decrease than increase, while positive E gives more increase than decrease.

The `umargin` control design block uses this model to represent gain and phase uncertainty in a feedback loop, setting α and E from the gain and phase margin values you specify when you create the block. The `diskmargin` command uses this model to compute disk-based gain and phase margins as discussed in the next section.

Disk Margins for SISO Loops

For a loop transfer L and a given eccentricity E, the `diskmargin` command finds the largest disk size α at which the closed-loop system `feedback(L*F,1)` is stable for all values of F. This value of α is called the *disk margin*. The disk-based gain margin `DGM` and disk-based phase margin `DPM` are the range of gain and phase variations represented by the corresponding disk.

For instance, compute the disk margin and associated disk-based gain and phase margins for a SISO transfer function, using the default $E = 0$.

```
E = 0;
L = tf(25,[1 10 10 10]);
DM = diskmargin(L,E);
alpha = DM.DiskMargin
```

```
alpha = 0.4581
```

```
DGM = DM.GainMargin
```

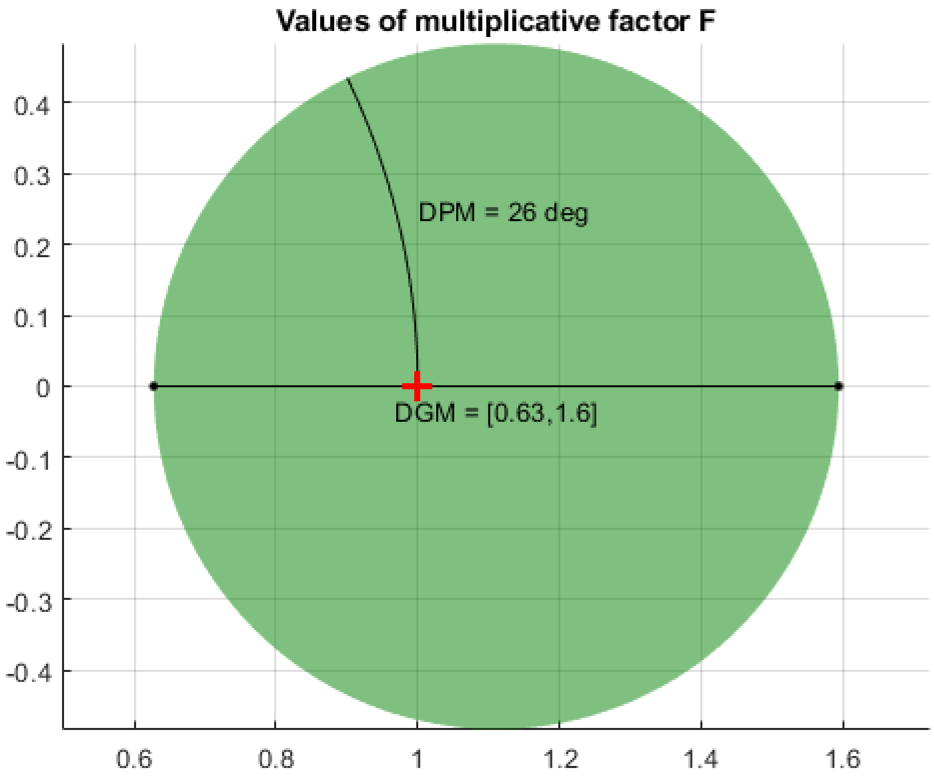
```
DGM = 1×2

    0.6273    1.5942
```

```
DPM = DM.PhaseMargin
```

```
DPM = 1×2

  -25.8017   25.8017
```

For this system, the balanced ($E = 0$) disk margin α is about 0.46. The corresponding disk-based gain margin DGM shows that the system remains stable for relative variations in gain between about 0.63 and 1.6, or for phase variations of about ±26 degrees. This result establishes stability for all values of F of in the disk:

```
diskmarginplot(DGM,'disk')
```

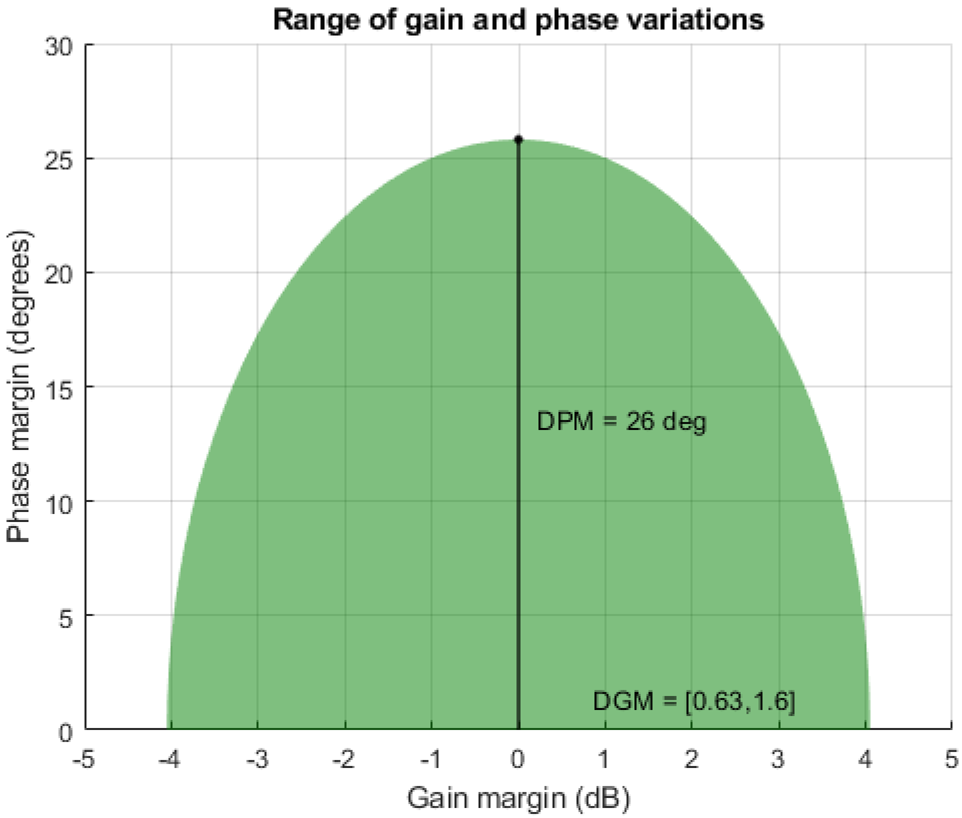


The gain margins are the intersection of the disk with the real axis. The phase margin is the largest angle between the real axis and a line through the origin tangent to the disk.

Combined Gain and Phase Variations

The gain margins DGM you obtain from diskmargin assume no phase variation, and the phase margins DPM assume no gain variation. In practice, your system can experience simultaneous gain and phase variations. `diskmarginplot` lets you visualize the ranges of simultaneous gain and phase variations that the system can tolerate.

```
diskmarginplot(DGM)
```



The shaded region shows the stable range of combined gain and phase variations. Thus, for instance, with no phase variation, the system can tolerate the full range `DGM` of gain variation, about –4 dB to 4 dB. If the phase is allowed to vary by ±17 degrees or so, the allowable gain variation drops to a range of about –3 dB to 3 dB.

Disk Margins and Eccentricity

The ranges shown above, computed for $E = 0$, represent a balanced gain variation, where `gmin` = 1/`gmax`. Varying the eccentricity can reveal whether the loop is more sensitive to gain increase or decrease. For example, using $E > 0$ may reveal that the feedback loop is very robust to gain increase, because positive E models more gain increase than decrease. This result, however, says little about robustness to gain decrease. Try computing the disk-based gain and phase margins for L, biasing the margins toward gain increase or gain decrease.

```
DMdec = diskmargin(L,-2);
DMinc = diskmargin(L,2);
DGMdec = DMdec.GainMargin
```

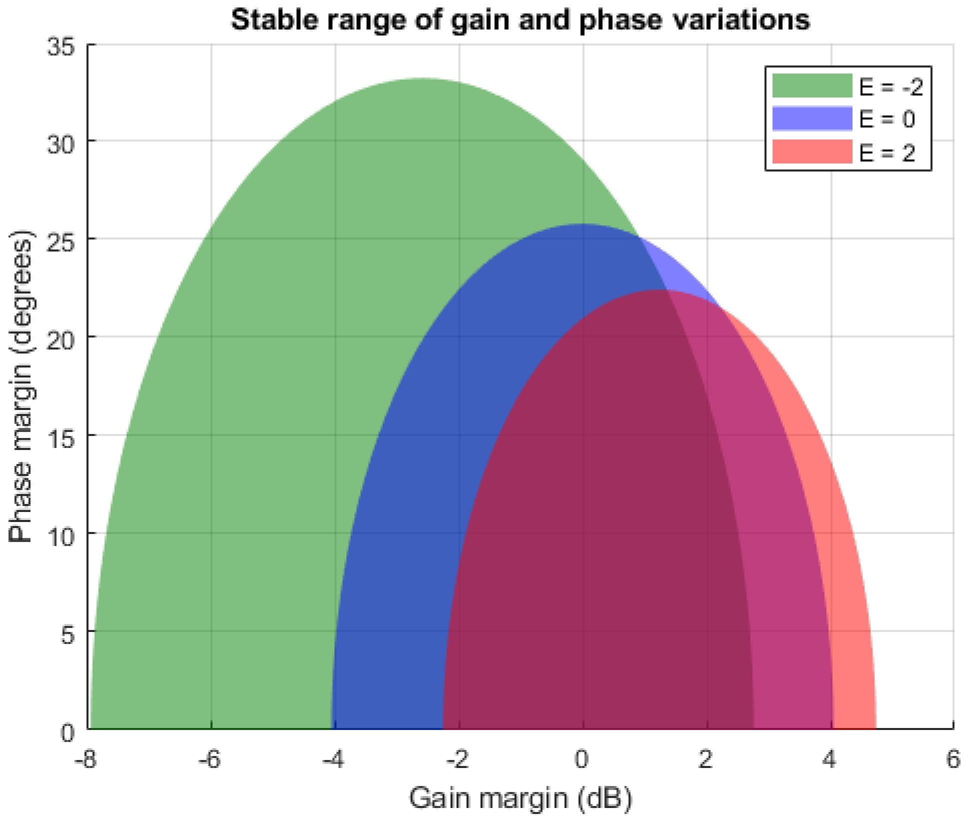
```
DGMdec = 1×2

    0.4013    1.3745
```

```
DGMinc = DMinc.GainMargin
```

```
DGMinc = 1×2

    0.7717    1.7247
```

Put together, these results show that in the absence of phase variation, stability is maintained for relative gain variations between 0.4 and 1.72. To see how the phase margin depends on these gain variations, plot the stable ranges of gain and phase variations for each `diskmargin` result.

```
diskmarginplot([DGMdec;DGM;DGMinc])
legend('E = -2','E = 0','E = 2')
title('Stable range of gain and phase variations')
```



This plot shows that the feedback loop can tolerate larger phase variations when the gain decreases. In other words, the loop stability is more sensitive to gain increase. Note that it would be misleading to just take the largest reported phase margin (nearly 30 degrees for E = –2). Indeed, this large value is predicated on a small gain increase of less than 3 dB. Since both gain and phase are subject to uncertainty in general, it is important to pay attention to combined variations. For example, the plot shows that when the gain increases by 4 dB, the phase margin drops to less than 15 degrees. By contrast, it remains greater than 30 degrees when the gain decreases by 4 dB.

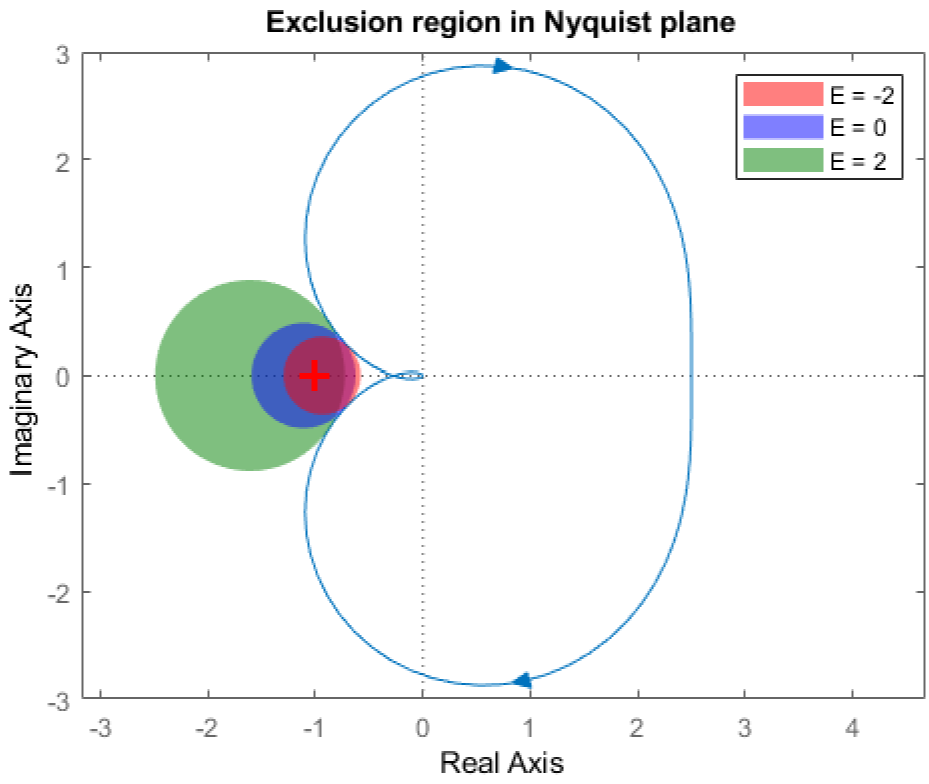
In conclusion, varying the eccentricity *E* can give a fuller picture of sensitivity to gain and phase uncertainty. Unless you are mostly concerned with gain variations in one direction (increase or decrease), it is not recommended to draw conclusions from a single nonzero value of *E*. Instead use the default *E* = 0 to get unbiased estimates of gain and phase margins. When using nonzero values of

E, use both positive and negative values to compare relative sensitivity to gain increase vs. gain decrease.

Nyquist Plot Interpretation of Disk Margin

The requirement of robust stability for the closed-loop system `feedback(L*F,1)` is equivalent to a requirement that `1 + L*F` ≠ 0. In the Nyquist plane, this requirement becomes $L(j\omega) \neq -1/F$. Thus the disk *F* for each value of *E* defines an exclusion region that the Nyquist curve does not enter if closed-loop stability is preserved. All such disks $-1/F$ contain the critical point (–1,0) and are tangent to the Nyquist curve. The eccentricity adjusts the size and position of the tangent disks, as illustrated in the following plot, which shows the exclusion regions for the three disk margins of *L* computed above. As *E* increases, each disk provides lower estimates of the classical gain and phase margins.

```
nyquist(L)
hold on
diskmarginplot([DGMdec;DGM;DGMinc],'nyquist')
p = findobj(gca,'type','patch');
legend(p,'E = -2','E = 0','E = 2')
hold off
```



MIMO Uncertainty Model and Disk Margins of MIMO Feedback Loops

For MIMO systems, the model applies an independent uncertainty disk F_j to each loop channel, given by

$$F_j = \frac{1 + \alpha[(1 - E)/2]\delta_j}{1 - \alpha[(1 + E)/2]\delta_j}.$$

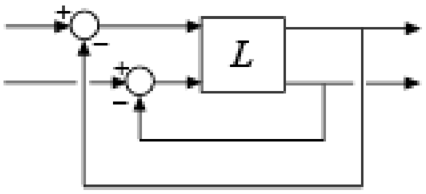
The model replaces the MIMO open-loop response *L* with *L*F*, where

$$F = \begin{pmatrix} F_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & F_N \end{pmatrix}.$$

Analogous to the SISO case, the disk margin is the largest value of α for which the closed-loop system `feedback(L*F,eye(N))` is stable for all values of *F*. It can be useful to consider independent variations across all feedback channels at once, as well as variations in individual channels. Therefore, `diskmargin` lets you compute:

- Loop-at-a-time margins — Maximum tolerable gain variations (or phase variations) in each feedback channel, computed with all other loops closed. Loop-at-a-time analysis effectively sets all δ_j to 0 for all channels except the channel under analysis.
- Multiloop margins — Maximum tolerable gain variations (or phase variations) across all feedback channels. Multiloop margins allow for independent variations in all feedback channels at the same time. The ability to capture such loop interactions is a key advantage of the disk-margin approach over classical margin analysis. Multiloop analysis typically yield smaller margins than loop-at-a-time analysis.

For instance, consider the 2-channel MIMO system of the following illustration.



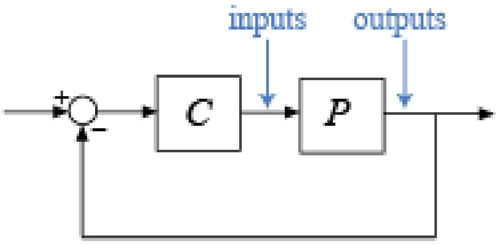
For this system, you can compute:

- Maximum tolerable gain variations (or phase variations) in the first channel (first system input to first system output)
- Maximum tolerable gain variations (or phase variations) in the second channel (second system input to second system output)
- Maximum tolerable independent gain variations (or phase variations) in both channels at the same time.

For details and examples of how to get these loop-at-a-time and multiloop margins, see `diskmargin`.

Variations at Plant Input or Plant Output

In some cases, the stability margins can vary depending on whether gain and phase variations are applied at the plant input or the plant output. `diskmargin` lets you compute margins for variations at the input, output, or both simultaneously. In general, the margins for simultaneous input and output variations are smaller than those for input or output only, and provide a more conservative guarantee of stability. Consider the SISO or MIMO closed-loop system of the following diagram.



You can compute the disk margins at the plant inputs and outputs as follows.

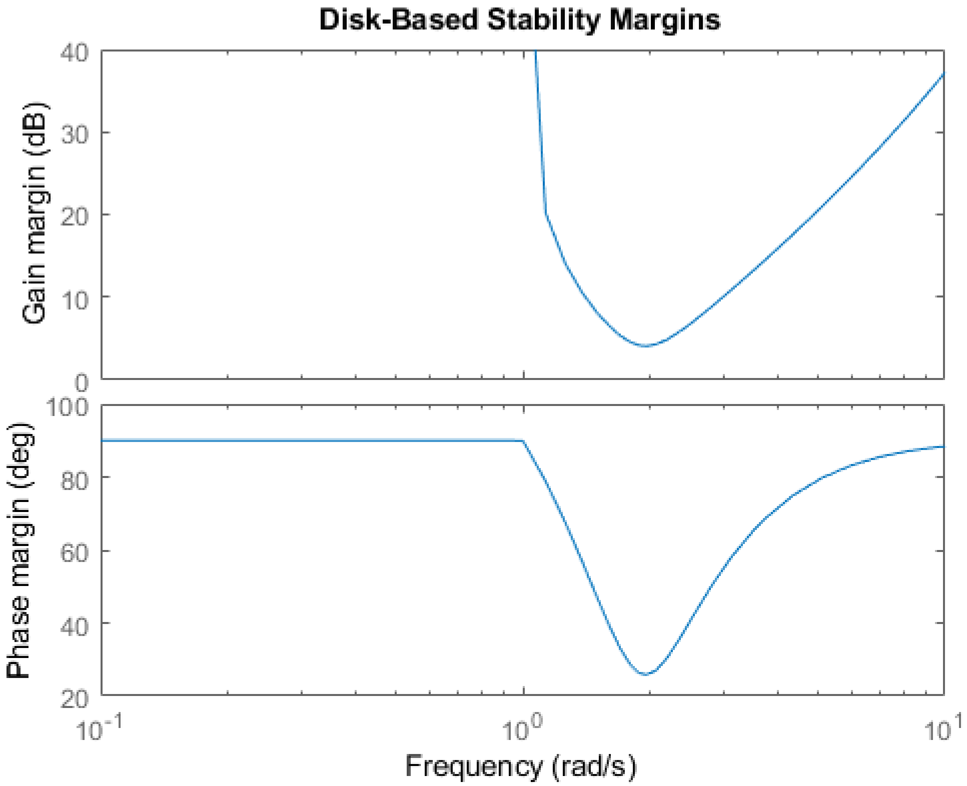
- `[DM,MM] = diskmargin(P*C)` returns the margins for variations at the plant outputs.
- `[DM,MM] = diskmargin(C*P)` returns the margins for variations at the plant inputs.
- `MMIO = diskmargin(P,C)` returns the margins for simultaneous variations at the plant outputs and inputs.

Frequency Dependence of Margins

In general, gain and phase margins vary across frequency. This variation is due to the frequency variation of the open-loop response L: For each frequency ω, there is a different largest $\alpha(\omega)$ such that $I + L(j\omega)F$ is invertible for all values of F in the disk. This value is the first $\alpha(\omega)$ for which the closed-loop pole crosses the $j\omega$ axis at the frequency ω, becoming unstable.

As the disk margin α varies across frequency, so does the corresponding tolerable range of gain or phase variations. Plotting these disk-based margins as a function of frequency provides information about frequency bands with weak margins.

```
L = tf(25,[1 10 10 10]);
diskmarginplot(L)
```



The disk margins returned by `diskmargin` are the minimum such values over all frequencies. Right-click on the plot generated by `diskmarginplot` for a data tip with information about these values.

Worst-Case Disk Margins

The disk margin is computed by applying an uncertainty to the nominal loop transfer *L* and computing how large that uncertainty can be while preserving closed-loop stability. If the loop transfer *L* is itself an uncertain system, then the disk margin also varies as a function of the other uncertainties in the system. The worst-case disk margin is the smallest disk margin that occurs within the ranges of the uncertainties modeled in *L*. It is also the minimum guaranteed margin over the uncertainty range.

Compute worst-case disk margins of an uncertain system using `wcdiskmargin`. This function estimates the worst-case disk margins and corresponding worst-case gain and phase margins for both loop-at-a-time and multiloop variations. The function also returns the worst-case perturbation, the combination of uncertain elements in *L* that yields the weakest margins. You can visualize worst-case disk-based margins with `wcdiskmarginplot`.

Disk Margins and Control System Tuning

Tuning With systune or Control System Tuner

The control system tuning tools in Control System Toolbox let you specify target gain and phase margins for loops in your tuned system. The tuning goals `TuningGoal.Margins` (for command-line tuning with `systune`) and Margins Goal (for tuning with Control System Tuner) use disk-based margins. Thus, when you specify independent gain and phase margins *GM* and *PM* for tuning, the software chooses the smallest α that enforces both values. This α is given by:

$$\alpha = \max\left[\frac{GM - 1}{GM + 1},\ \tan(PM/2)\right].$$

In applying this value of α, the tuning software assumes $E = 0$.

Visualizing margin goals using `viewGoal` or Control System Tuner tuning-goal plots is equivalent to `diskmarginplot(L)`, where `L` is the tuned open-loop response.

Robust Design With musyn

When you perform robust controller tuning with `musyn`, you can model gain and phase variations directly in your system using `umargin`. Then, performing robust controller design with `musyn` enforces robust stability for the modeled range of gain and phase variations. This approach is useful because it allows you to study the effects of the expected gain and phase variations on all aspects of system performance using the same model you use for tuning. For an example, see “Robust Controller for Spinning Satellite” on page 3-102.

A disadvantage of this approach is that `musyn` does not only enforce robust stability over the entire modeled uncertainty range. It also attempts to enforce robust performance. (See “Robust Performance Measure for Mu Synthesis” on page 3-11.) Achieving this more stringent requirement is typically impossible or results in intolerable degradation of nominal performance. Thus, you might need to reduce the modeled gain and phase variations to maintain reasonable performance. `systune` and Control System Tuner do not have this drawback because they handle margin goals independently of any performance goals.

References

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

See Also

`diskmargin` | `diskmarginplot` | `umargin` | `wcdiskmargin`

More About

- “Robustness and Worst-Case Analysis” on page 2-21

MIMO Stability Margins for Spinning Satellite

This example shows that in MIMO feedback loops, disk margins are more reliable stability margin estimates than the classical, loop-at-a-time gain and phase margins. It first appeared in Reference [1].

Spinning Satellite

This example is motivated by controlling the transversal angular rates in a spinning satellite (see Reference [2]). The cylindrical body spins around its axis of symmetry (*z* axis) with constant angular rate Ω. The goal is to independently control the angular rates ω_x and ω_y around the *x* and *y* axes using torques u_x and u_y. The equations of motion lead to the second-order, two-input, two-output model:

$$\begin{pmatrix} \dot{\omega}_x \\ \dot{\omega}_y \end{pmatrix} = \begin{pmatrix} 0 & a \\ -a & 0 \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \end{pmatrix} + \begin{pmatrix} u_x \\ u_y \end{pmatrix}$$

$$\begin{pmatrix} \nu_1 \\ \nu_2 \end{pmatrix} = \begin{pmatrix} 1 & a \\ -a & 1 \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \end{pmatrix}, \quad a = 10\,.$$

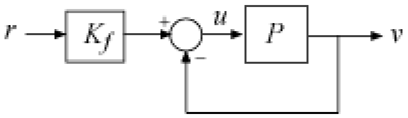
Consider a static control law that combines feedforward and unit-feedback actions:

$u = K_f r - \nu.$

Setting the feedforward gain as follows achieves perfect decoupling and makes each loop respond as a first-order system with unit time constant.

$$K_f = \frac{1}{1 + a^2} \begin{pmatrix} 1 & -a \\ a & 1 \end{pmatrix}$$

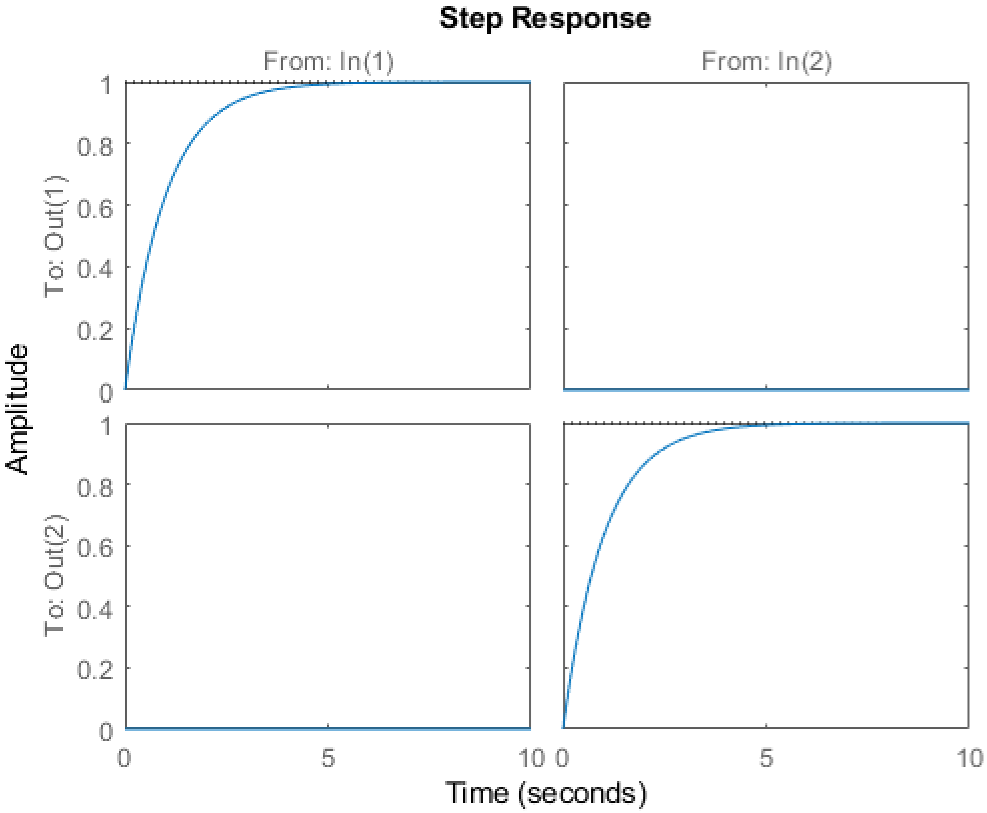
The resulting system is the two-loop control system of the following diagram.



Create a state-space model of the satellite and the controller to see the performance of the system with this controller.

```
% Plant
a = 10;
A = [0 a;-a 0];
B = eye(2);
C = [1 a;-a 1];
D = 0;
P = ss(A,B,C,D);

% Prefilter
Kf = [1 -a;a 1]/(1+a^2);

% Closed-loop model
```

```
T = feedback(P,eye(2))*Kf;
step(T,10)
```



Loop-at-a-Time Gain and Phase Margins

The classical gain and phase margins are a customary way to gauge the robustness of SISO feedback loops to plant uncertainty. In MIMO control systems, the gain and phase margins are often assessed one loop at a time, meaning that the margins are computed independently for each feedback channel with the other loops closed. Robustness is clearly in doubt when one loop is found to have weak margins. Unfortunately, the converse is not true. Each individual loop can have strong gain and phase margins while overall robustness is weak. This is because the loop-at-a-time approach fails to account for loop interactions and conditioning of the MIMO loop transfer. The spinning satellite example is a perfect illustration of this point.

To compute the loop-at-a-time gain and phase margins, introduce an analysis point at the plant input to facilitate access to the open-loop responses.

```
uAP = AnalysisPoint('u',2);
T = feedback(P*uAP,eye(2))*Kf;
```

Use `getLoopTransfer` to access the SISO loop transfers at u_x with the y loop closed, and at u_y with the x loop closed.

```
Lx = getLoopTransfer(T,'u(1)',-1);
Ly = getLoopTransfer(T,'u(2)',-1);
```

Finally, use `margin` to compute the loop-at-a-time classical gain and phase margins.

```
[gmx,pmx] = margin(Lx)
```

```
gmx = Inf
```

```
pmx = 90
```

```
[gmy,pmy] = margin(Ly)
```

```
gmy = Inf
```

```
pmy = 90
```

`margin` reports a gain margin of `Inf` and phase margin of 90°, as stable against variations in gain and phase as a SISO loop can be. This result is not surprising given that the loop transfers `Lx` and `Ly` are pure integrators.

```
tf(Lx), tf(Ly)
```

```
ans =
 
  From input "u(1)" to output "u(1)":
  1
  -
  s
 
Continuous-time transfer function.


ans =
 
  From input "u(2)" to output "u(2)":
  1
  -
  s
 
Continuous-time transfer function.
```

Despite this loop-at-a-time result, the MIMO feedback loop T can be destabilized by a 10% gain change affecting both loops. To demonstrate this destabilization, multiply the plant by a static gain that decreases the gain by 10% in the u_x channel and increases it by 10% in the u_y channel. Then compute the poles of the unit feedback loop to see that one of them has moved into the right half-plane, driving the loop unstable.

```
dg = diag([1-0.1,1+0.1]);
pole(feedback(P*dg,eye(2)))
```

```
ans = 2×1

   -2.0050
    0.0050
```

Because model uncertainty typically affects all feedback loops, the loop-at-a-time margins tend to overestimate actual robustness and can miss important issues with MIMO designs.

Disk Margins

The notion of *disk margin* gives a more comprehensive assessment of robustness in MIMO feedback loops by explicitly modeling and accounting for loop interactions. See “Stability Analysis Using Disk

Margins” on page 2-2 for additional background information. To compute the SISO and MIMO disk margins for the spinning satellite example, extract the MIMO loop transfer `L` and use the `diskmargin` command.

```
L = getLoopTransfer(T,'u',-1);
[DM,MM] = diskmargin(L);
```

The first output `DM` contains the loop-at-a-time disk-based margins. These are lower estimates of the classical gain and phase margins. For this model, the disk-based margins also report a gain margin of `Inf` and phase margin of 90° for each individual loop. So far nothing new.

```
DM(1)
```

```
ans = struct with fields:
           GainMargin: [0 Inf]
          PhaseMargin: [-90 90]
           DiskMargin: 2
           LowerBound: 2
           UpperBound: 2
            Frequency: 0
    WorstPerturbation: [2x2 ss]
```

```
DM(2)
```

```
ans = struct with fields:
           GainMargin: [0 Inf]
          PhaseMargin: [-90 90]
           DiskMargin: 2
           LowerBound: 2
           UpperBound: 2
            Frequency: 0
    WorstPerturbation: [2x2 ss]
```

The second output `MM` contains the MIMO disk-based margins.

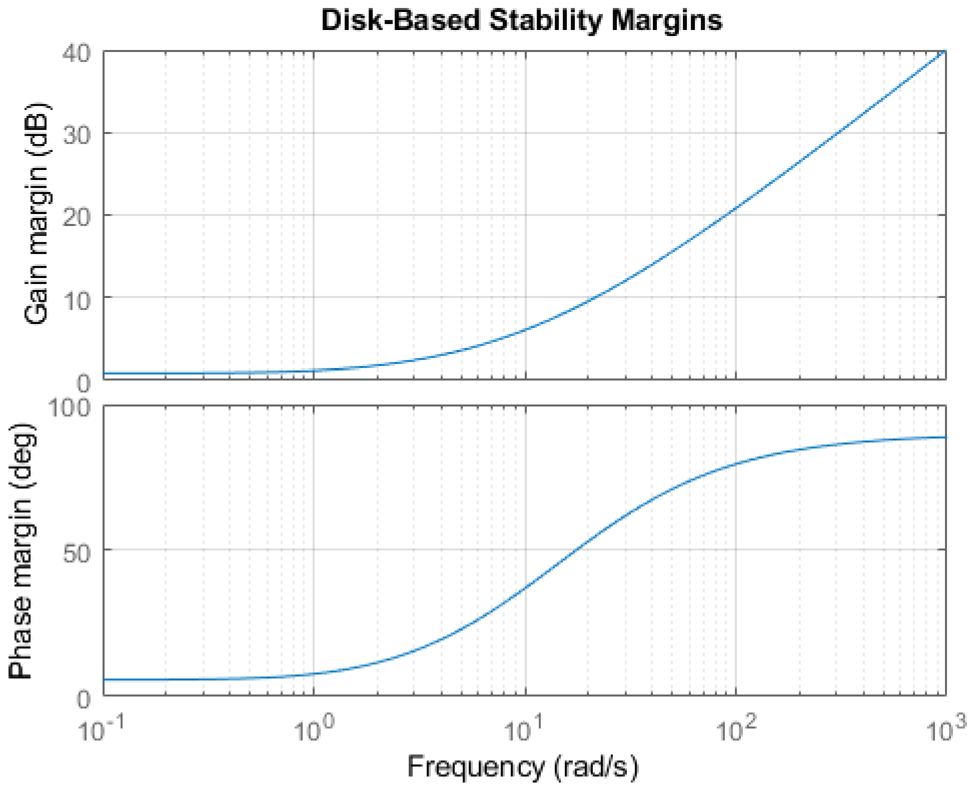
```
MM
```

```
MM = struct with fields:
           GainMargin: [0.9051 1.1049]
          PhaseMargin: [-5.7060 5.7060]
           DiskMargin: 0.0997
           LowerBound: 0.0997
           UpperBound: 0.0999
            Frequency: 1.0000e-04
    WorstPerturbation: [2x2 ss]
```

This multiloop disk-based margin is the most reliable measure of MIMO robustness since it accounts for independent gain and phase variations in all loop channels at the same time. Here `MM` reports a gain margin of [0.905,1.105], meaning that the open-loop gain can increase or decrease by a factor of 1.105 while preserving closed-loop stability. The phase margin is 5.7°. This result is in line with the destabilizing perturbation `dg` above which corresponds to a relative gain change in the range [0.90,1.10].

The `diskmarginplot` command lets you visualize the disk-based margins as a function of frequency. For this system, that the disk-based margins are weakest near DC (zero frequency).

```
clf
diskmarginplot(L)
grid
```



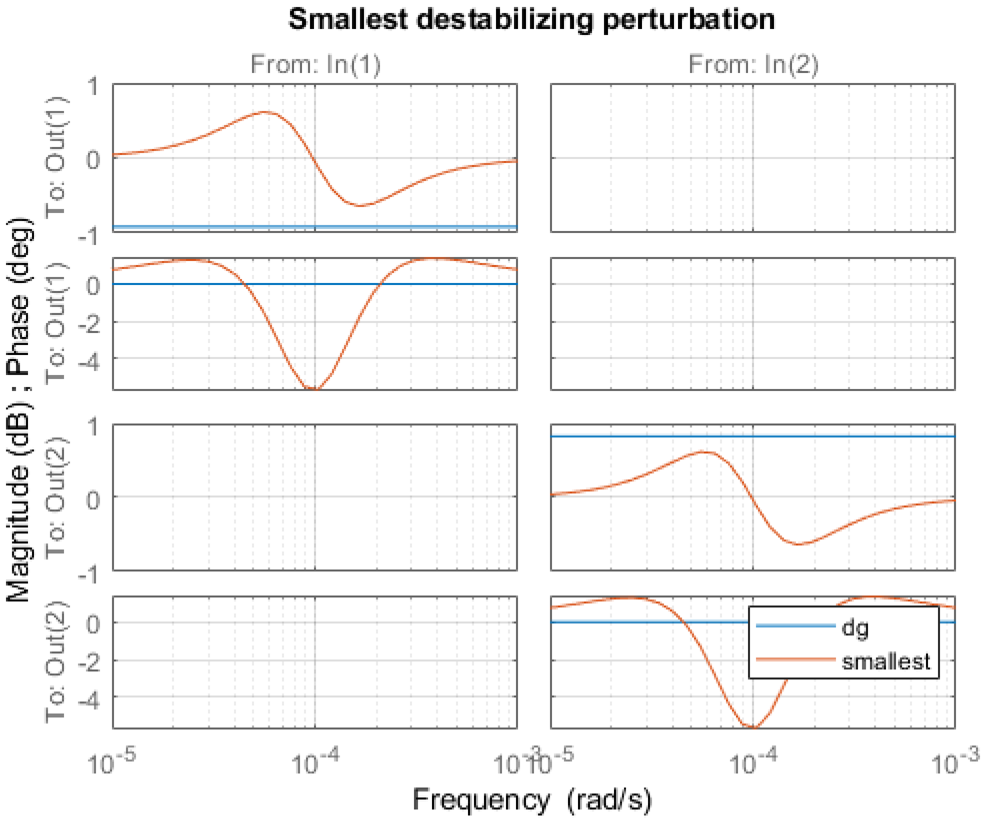
The struct MM also contains the smallest destabilizing perturbation MM.WorstPerturbation. This dynamic perturbation is a realization of the smallest destabilizing change in gain and phase. Note that this perturbation modifies both gain and phase in both feedback channels.

```
WGP = MM.WorstPerturbation;
pole(feedback(P*WGP,eye(2)))
```

```
ans = 8×1 complex

  -1.0000 + 0.0001i
  -1.0000 - 0.0001i
  -0.0002 + 0.0002i
  -0.0002 - 0.0002i
   0.0000 + 0.0001i
   0.0000 - 0.0001i
  -0.0000 + 0.0000i
  -0.0000 - 0.0000i
```

Compare the dynamic perturbation WGP with the static gain dg used above to destabilize the system.

```
bode(ss(dg),WGP), grid
title('Smallest destabilizing perturbation')
legend('dg','smallest')
```



Conclusion

For multi-loop control systems, disk margins are a more reliable robust stability test than loop-at-a-time gain and phase margins. The `diskmargin` function computes both loop-at-a-time and MIMO disk margins and the corresponding smallest destabilizing perturbations. You can use these perturbations for further analysis and risk assessment, for example, by injecting them in a detailed nonlinear simulation of the system.

To learn how to design a controller with adequate stability margins for this plant, see the example “Robust Controller for Spinning Satellite” on page 3-102.

References

[1] Doyle, J. “Robustness of Multiloop Linear Feedback Systems.” In *1978 IEEE Conference on Decision and Control Including the 17th Symposium on Adaptive Processes*, 12–18. San Diego, CA, USA: IEEE, 1978. https://doi.org/10.1109/CDC.1978.267885.

[2] Zhou, K., J.C.Doyle, and K. Glover. *Robust and Optimal Control*, Englewood Cliffs, NJ: Prentice-Hall, 1996, Section 9.6.

See Also

`diskmargin` | `diskmarginplot`

More About

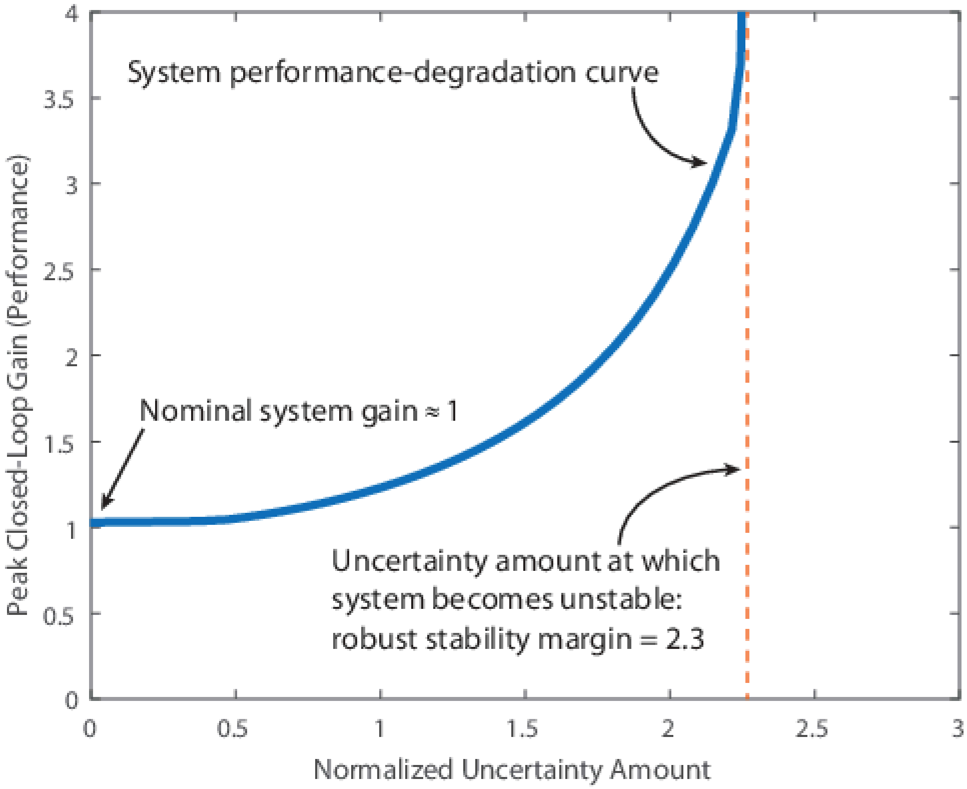
- “Stability Analysis Using Disk Margins” on page 2-2
- “Robust Controller for Spinning Satellite” on page 3-102

Robustness and Worst-Case Analysis

In robust control design, performance is expressed and measured in terms of the peak gain (the H_∞ norm or peak singular value) of a system. The smaller this gain is, the better the system performance. The performance of a nominally stable uncertain system generally degrades as the amount of uncertainty increases. Use robustness analysis and worst-case analysis to examine how the amount of uncertainty in your system affects the stability and peak gain of the system.

Robustness Analysis

Robustness analysis is about finding the maximum amount of uncertainty compatible with stability or with a given performance level. The following illustration shows a typical trade-off curve between performance and robustness. Here, the peak gain (peak magnitude on a Bode plot or singular-value plot) characterizes the system performance.



The *x*-axis quantifies the normalized amount of uncertainty. The value *x* = 1 corresponds to the uncertainty ranges specified in the model. *x* = 2 represents the system with twice as much uncertainty. *x* = 0 corresponds to the nominal system. (See `actual2normalized` for more details about normalized uncertainty ranges.) The *y*-axis is performance, measured as the peak gain of some closed-loop transfer function. For instance, if the closed-loop transfer function measures the sensitivity of an error signal to some disturbance, then higher peak gain corresponds to poorer disturbance rejection.

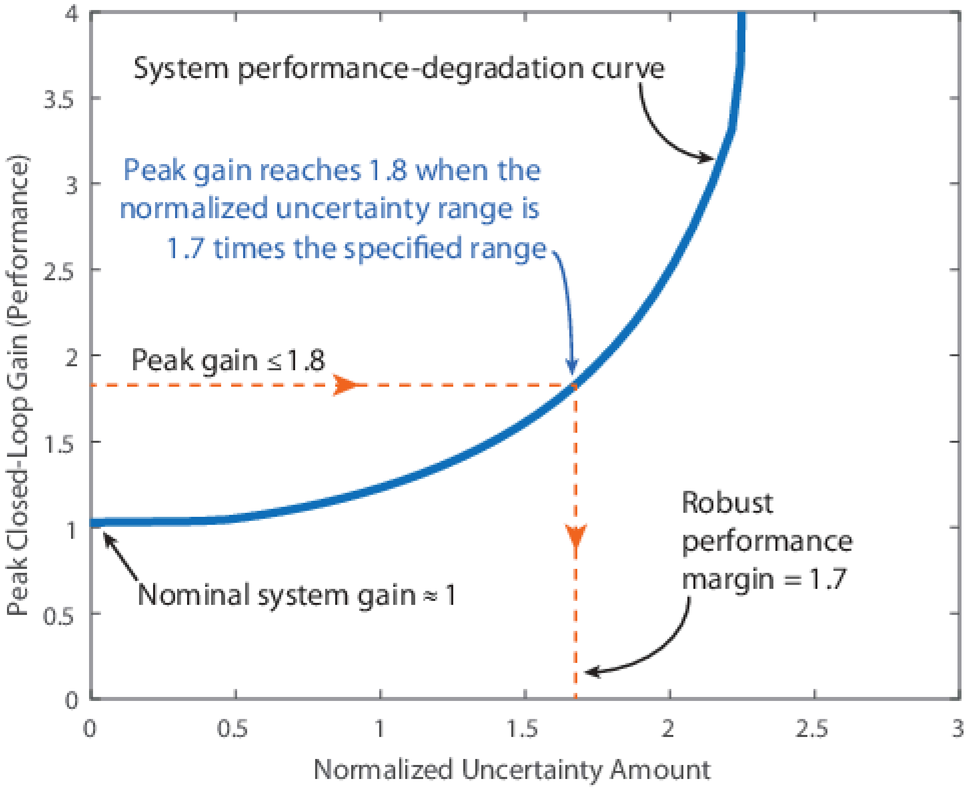
When all uncertain elements are set to their nominal values (*x* = 0), the gain of the system is its nominal value. In the figure, the nominal system gain is about 1. As the range of values that the uncertain elements can take increases, the peak gain over the uncertainty range increases. The heavy blue line represents the peak gain, and is called the system performance degradation curve. It increases monotonically as a function of the uncertainty amount.

Robust Stability Margin

The system performance degradation curve typically has a vertical asymptote corresponding to the robust stability margin. This margin is the maximum amount of uncertainty that the system can tolerate while remaining stable. For the system of the previous illustration, the peak gain becomes infinite at around $x = 2.3$. In other words, the system becomes unstable when the uncertainty range is 2.3 times that specified in the model (in normalized units). Therefore, the robust stability margin is 2.3. To compute the robust stability margin for an uncertain system model, use the `robstab` function.

Robust Performance Margin

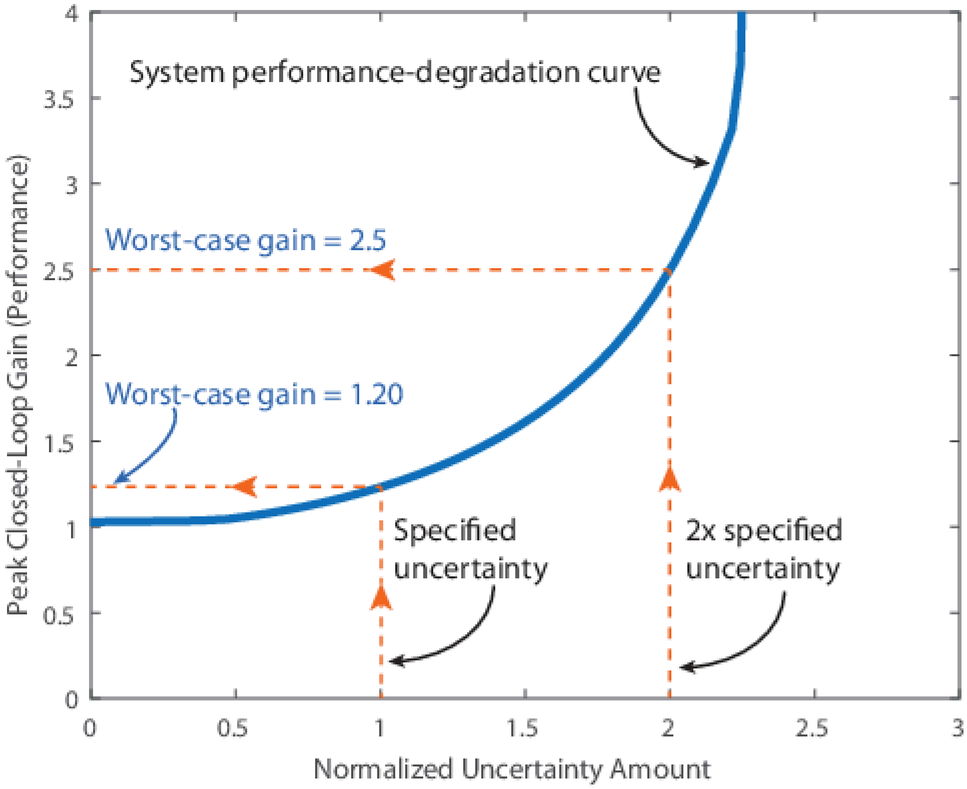
The robust performance margin for a given gain, γ, is the maximum amount of uncertainty the system can tolerate while having a peak gain less than γ. For example, in the following illustration, suppose that you want to keep the peak closed-loop gain below 1.8. For that peak gain, the robust performance margin is about 1.7. This value means that the peak gain of the system remains below 1.8 as long as the uncertainty remains within 1.7 times the specified uncertainty (in normalized units).



To compute the robust performance margin for an uncertain system model, use the `robgain` function.

Worst-Case Gain Measure

The worst-case gain is the largest value that the peak gain can take over a specific uncertainty range. This value is the counterpart of the robust performance margin. While the robust performance margin measures the maximum amount of uncertainty compatible with a particular peak gain level, the worst-case gain measures the maximum gain associated with a particular uncertainty amount. For instance, in the following illustration, the worst-case gain for the uncertainty amount specified in the model is about 1.20. If that uncertainty amount is doubled, the worst-case gain increases to 2.5.



To compute the worst-case gain for an uncertain system model, use the `wcgain` function. The `ULevel` option of the `wcOptions` command lets you compute the worst-case gain for different amounts of uncertainty.

See Also

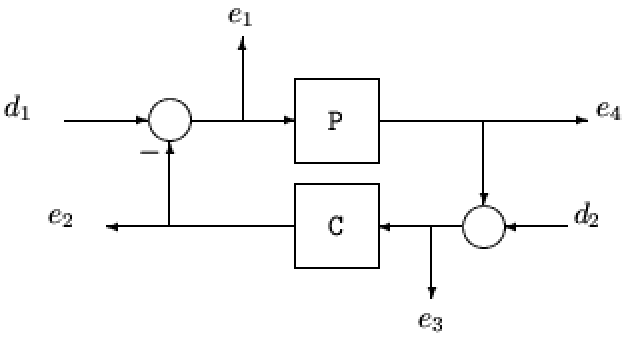
`robgain` | `robstab` | `wcgain`

Related Examples

- “Robust Stability and Worst-Case Gain of Uncertain System”
- “Worst-Case Sensitivity and Complementary Sensitivity” on page 2-24
- “MIMO Robustness Analysis”
- “Stability Analysis Using Disk Margins” on page 2-2

Worst-Case Sensitivity Functions of Feedback Loops

The sensitivity function and the complementary sensitivity function are two transfer functions related to the robustness and performance of a closed-loop system. Consider a general multivariable closed-loop control structure, as in the following illustration.



The following table gives the values of the input and output sensitivity functions for this control structure.

Description	Equation
Input sensitivity S_i (closed-loop transfer function from d_1 to e_1)	$S_i = (I + CP)^{-1}$
Input complementary sensitivity T_i (closed-loop transfer function from d_1 to e_2)	$T_i = CP(I + CP)^{-1}$
Output sensitivity S_o (closed-loop transfer function from d_2 to e_2)	$S_o = (I + PC)^{-1}$
Output complementary sensitivity T_o (closed-loop transfer function from d_2 to e_4)	$T_o = PC(I + PC)^{-1}$
Input loop transfer function L_i	$L_i = CP$
Output loop transfer function L_o	$L_o = PC$

Worst-Case Sensitivity and Complementary Sensitivity

When you have an uncertain plant model and a controller model, you can compute the worst-case sensitivity and complementary sensitivity functions for robustness analysis. To do so, construct the transfer function you want to evaluate and use `wcgain` to find the perturbations that yield the worst-case gain for that transfer function. Then, use `usubs` to compute the transfer function corresponding to that worst-case gain.

For this example, create a SISO uncertain plant P and a PID controller.

```
delta = ultidyn('delta',[1 1]);
tau = ureal('tau',5,'range',[4 6]);
P = tf(1,[tau 1])*(1+0.25*delta);
C = pid(4,4);
```

Construct the uncertain sensitivity and complementary sensitivity transfer functions, $S_i = (I + \mathrm{CP})^{-1}$ and $T_i = I - S_i$, respectively. (For this SISO system, the input and output sensitivity functions are equal.)

```
Si = feedback(1,C*P);
Ti = 1 - Si;
```

Compute the worst-case peak gains of `Si` and `Ti` and the corresponding worst-case perturbations using `wcgain`.

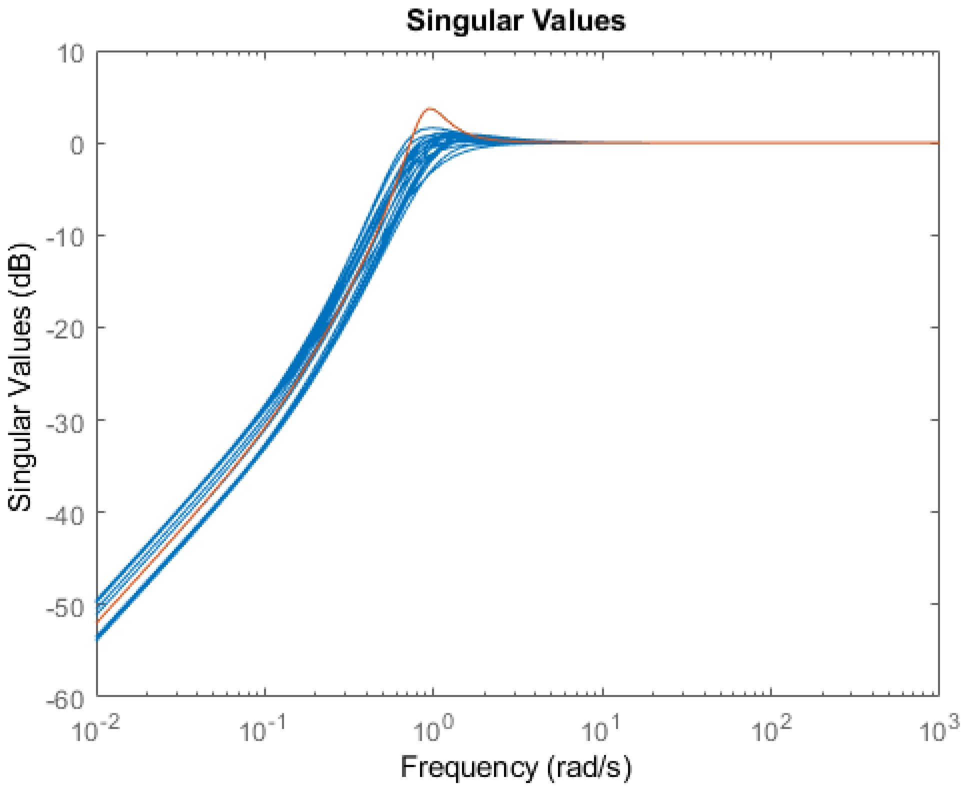
```
[wcgS,wcuS] = wcgain(Si);
[wcgT,wcuT] = wcgain(Ti);
```

Finally, evaluate the sensitivity functions with these worst-case perturbations.

```
Siwc = usubs(Si,wcuS);
Tiwc = usubs(Ti,wcuT);
```

`Siwc` and `Tiwc` are the worst-case sensitivity and complementary sensitivity functions for the uncertainty specified in the plant. Examine the effect of uncertainty on the sensitivity function `Si` by plotting the frequency response of some samples. The actual worst-case peak gain `wcgS` can be significantly higher than the random samples show.

```
rng(0); % for reproducibility
sigma(Si,Siwc)
```



See Also

`wcgain`

More About

- “Robustness and Worst-Case Analysis” on page 2-21

Robust Stability, Robust Performance and Mu Analysis

This example shows how to use Robust Control Toolbox™ to analyze and quantify the robustness of feedback control systems. It also provides insight into the connection with mu analysis and the `mussv` function.

System Description

Figure 1 shows the block diagram of a closed-loop system. The plant model P is uncertain and the plant output y must be regulated to remain small in the presence of disturbances d and measurement noise n.

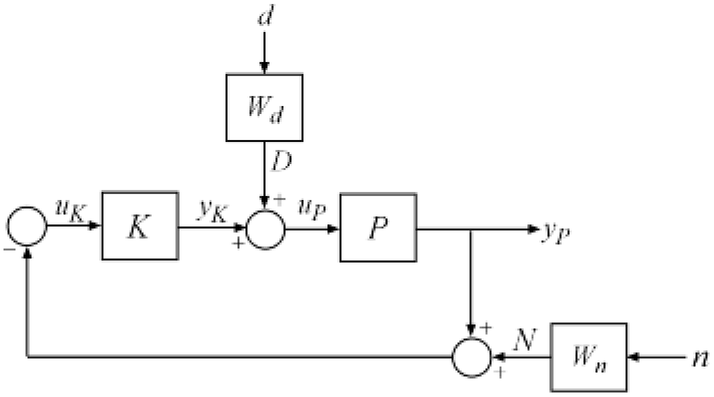


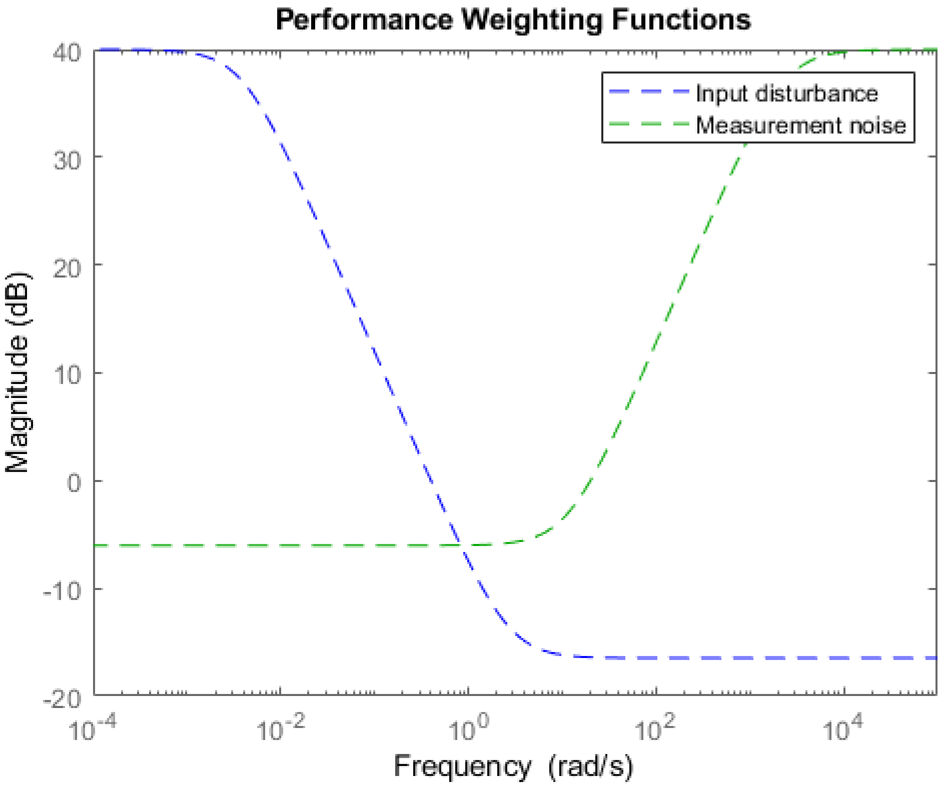
Figure 1: Closed-loop system for robustness analysis

Disturbance rejection and noise insensitivity are quantified by the performance objective

$$\left\|\left(P(1+KP)^{-1}W_d, (1+PK)^{-1}W_n\right)\right\|_\infty$$

where W_d and W_n are weighting functions reflecting the frequency content of d and n. Here W_d is large at low frequencies and W_n is large at high frequencies.

```
Wd = makeweight(100,.4,.15);
Wn = makeweight(0.5,20,100);
bodemag(Wd,'b--',Wn,'g--')
title('Performance Weighting Functions')
legend('Input disturbance','Measurement noise')
```



Creating an Uncertain Plant Model

The uncertain plant model P is a lightly-damped, second-order system with parametric uncertainty in the denominator coefficients and significant frequency-dependent unmodeled dynamics beyond 6 rad/s. The mathematical model looks like:

$$P(s) = \frac{16}{s^2 + 0.16s + k}(1 + W_u(s)\delta(s))$$

The parameter k is assumed to be about 40% uncertain, with a nominal value of 16. The frequency-dependent uncertainty at the plant input is assumed to be about 30% at low frequency, rising to 100% at 10 rad/s, and larger beyond that. Construct the uncertain plant model P by creating and combining the uncertain elements:

```
k = ureal('k',16,'Percentage',30);
delta = ultidyn('delta',[1 1],'SampleStateDim',4);
Wu = makeweight(0.3,10,20);
P = tf(16,[1 0.16 k]) * (1+Wu*delta);
```

Designing a Controller

We use the controller designed in the example "Improving Stability While Preserving Open-Loop Characteristics". The plant model used there happens to be the nominal value of the uncertain plant model created above. For completeness, we repeat the commands used to generate the controller.

```
K_PI = pid(1,0.8);
K_rolloff = tf(1,[1/20 1]);
```

```
Kprop = K_PI*K_rolloff;
[negK,~,Gamma] = ncfsyn(P.NominalValue,-Kprop);
K = -negK;
```

Closing the Loop

Use connect to build an uncertain model of the closed-loop system of Figure 1. Name the signals coming in and out of each block and let connect do the wiring:

```
P.u = 'uP';  P.y = 'yP';
K.u = 'uK';  K.y = 'yK';
S1 = sumblk('uP = yK + D');
S2 = sumblk('uK = -yP - N');
Wn.u = 'n'; Wn.y = 'N';
Wd.u = 'd'; Wd.y = 'D';
ClosedLoop = connect(P,K,S1,S2,Wn,Wd,{'d','n'},'yP');
```

The variable ClosedLoop is an uncertain system with two inputs and one output. It depends on two uncertain elements: a real parameter k and an uncertain linear, time-invariant dynamic element delta.

```
ClosedLoop
```

```
ClosedLoop =

  Uncertain continuous-time state-space model with 1 outputs, 2 inputs, 10 states.
  The model uncertainty consists of the following blocks:
    delta: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    k: Uncertain real, nominal = 16, variability = [-30,30]%, 1 occurrences

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

Robust Stability Analysis

The classical margins from allmargin indicate good stability robustness to unstructured gain/phase variations within the loop.

```
allmargin(P.NominalValue*K)
```

```
ans = struct with fields:
     GainMargin: [1.4860e-15 6.3299 11.1423]
    GMFrequency: [0 1.6110 15.1667]
    PhaseMargin: [80.0276 -99.6641 63.7989]
    PMFrequency: [0.4472 3.1460 5.2319]
    DelayMargin: [3.1236 1.4443 0.2128]
    DMFrequency: [0.4472 3.1460 5.2319]
         Stable: 1
```

Does the closed-loop system remain stable for all values of k, delta in the ranges specified above? Answering this question requires a more sophisticated analysis using the robstab function.

```
[stabmarg,wcu] = robstab(ClosedLoop);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 1.4671
           UpperBound: 1.4701
```

```
    CriticalFrequency: 5.8542
```

The variable `stabmarg` gives upper and lower bounds on the **robust stability margin**, a measure of how much uncertainty on `k`, `delta` the feedback loop can tolerate before becoming unstable. For example, a margin of 0.8 indicates that as little as 80% of the specified uncertainty level can lead to instability. Here the margin is about 1.5, which means that the closed loop will remain stable for up to 150% of the specified uncertainty.

The variable `wcu` contains the combination of `k` and `delta` closest to their nominal values that causes instability.

```
wcu
```

```
wcu = struct with fields:
    delta: [1x1 ss]
        k: 23.0563
```

We can substitute these values into `ClosedLoop` and verify that these values cause the closed-loop system to be unstable.

```
format short e
pole(usubs(ClosedLoop,wcu))
```

Note that the natural frequency of the unstable closed-loop pole is given by `stabmarg.CriticalFrequency`:

```
stabmarg.CriticalFrequency
```

```
ans =
   5.8542e+00
```

Connection with Mu Analysis

The structured singular value, or μ, is the mathematical tool used by `robstab` to compute the robust stability margin. If you are comfortable with structured singular value analysis, you can use the `mussv` function directly to compute mu as a function of frequency and reproduce the results above. The function `mussv` is the underlying engine for all robustness analysis commands.

To use `mussv`, we first extract the `(M,Delta)` decomposition of the uncertain closed-loop model `ClosedLoop`, where `Delta` is a block-diagonal matrix of (normalized) uncertain elements. The 3rd output argument of `lftdata`, `BlkStruct`, describes the block-diagonal structure of `Delta` and can be used directly by `mussv`

```
[M,Delta,BlkStruct] = lftdata(ClosedLoop);
```

For robust stability analysis, only the channels of `M` associated with the uncertainty channels are used. Based on the row/column size of `Delta`, select the proper columns and rows of `M`. Remember that the rows of `Delta` correspond to the columns of `M`, and vice versa. Consequently, the column dimension of `Delta` is used to specify the rows of `M`:

```
szDelta = size(Delta);
M11 = M(1:szDelta(2),1:szDelta(1));
```

In its simplest form, mu-analysis is performed on a finite grid of frequencies. Pick a vector of logarithmically-spaced frequency points and evaluate the frequency response of M11 over this frequency grid.

```
omega = logspace(-1,2,50);
M11_g = frd(M11,omega);
```

Compute mu(M11) at these frequencies and plot the resulting lower and upper bounds:

```
mubnds = mussv(M11_g,BlkStruct,'s');

LinMagopt = bodeoptions;
LinMagopt.PhaseVisible = 'off'; LinMagopt.XLim = [1e-1 1e2]; LinMagopt.MagUnits = 'abs';
bodeplot(mubnds(1,1),mubnds(1,2),LinMagopt);
xlabel('Frequency (rad/sec)');
ylabel('Mu upper/lower bounds');
title('Mu plot of robust stability margins (inverted scale)');
```

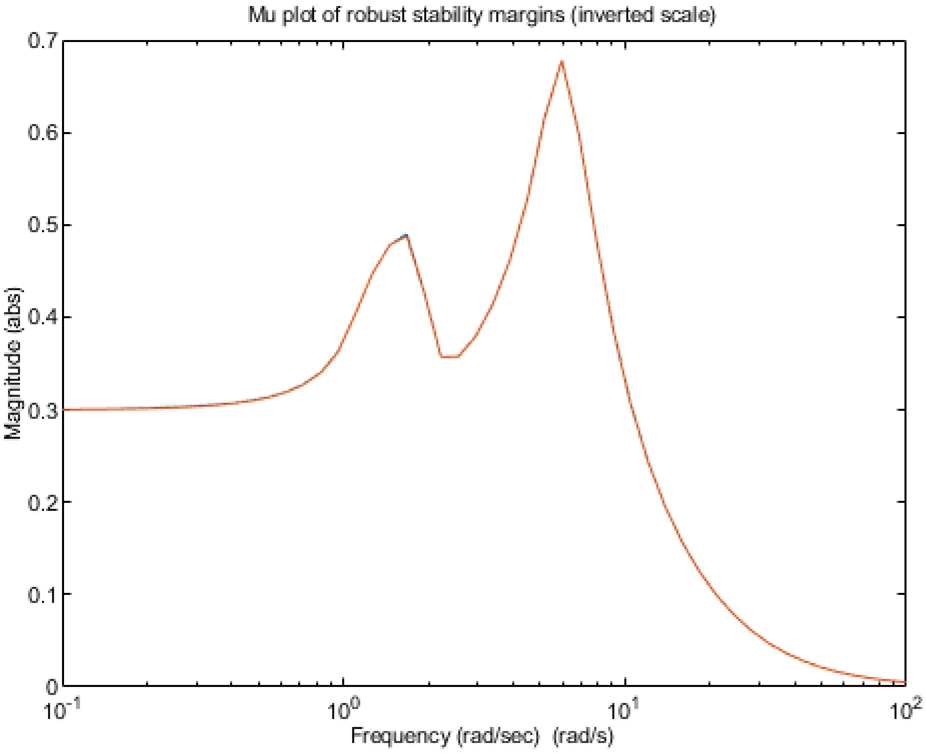


Figure 3: Mu plot of robust stability margins (inverted scale)

The robust stability margin is the reciprocal of the structured singular value. Therefore upper bounds from mussv become lower bounds on the stability margin. Make these conversions and find the destabilizing frequency where the mu upper bound peaks (that is, where the stability margin is smallest):

```
[pkl,wPeakLow] = getPeakGain(mubnds(1,2));
[pku] = getPeakGain(mubnds(1,1));
```

```
SMfromMU.LowerBound = 1/pku;
SMfromMU.UpperBound = 1/pkl;
SMfromMU.CriticalFrequency = wPeakLow;
```

Compare SMfromMU to the bounds stabmarg computed with robstab. The values are in rough agreement with robstab yielding slightly weaker margins. This is because robstab uses a more sophisticated approach than frequency gridding and can accurately compute the peak value of mu across frequency.

```
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 1.4671e+00
           UpperBound: 1.4701e+00
    CriticalFrequency: 5.8542e+00
```

```
SMfromMU
```

```
SMfromMU = struct with fields:
           LowerBound: 1.4748e+00
           UpperBound: 1.4748e+00
    CriticalFrequency: 5.9636e+00
```

Robust Performance Analysis

For the nominal values of the uncertain elements k and delta, the closed-loop gain is less than 1:

```
getPeakGain(ClosedLoop.NominalValue)
```

```
ans =
   9.8040e-01
```

This says that the controller K meets the disturbance rejection and noise insensitivity goals. But is this nominal performance maintained in the face of the modeled uncertainty? This question is best answered with robgain.

```
opt = robOptions('Display','on');
[perfmarg,wcu] = robgain(ClosedLoop,1,opt);
```

```
Computing peak...  Percent completed: 100/100
The performance level 1 is not robust to the modeled uncertainty.
 -- The gain remains below 1 for up to 39.9% of the modeled uncertainty.
 -- There is a bad perturbation amounting to 39.9% of the modeled uncertainty.
 -- This perturbation causes a gain of 1 at the frequency 0.129 rad/seconds.
```

The answer is negative: robgain found a perturbation amounting to only 40% of the specified uncertainty that drives the closed-loop gain to 1.

```
getPeakGain(usubs(ClosedLoop,wcu),1e-6)
```

```
ans =
   1.0000e+00
```

This suggests that the closed-loop gain will exceed 1 for 100% of the specified uncertainty. This is confirmed by computing the worst-case gain:

```
wcg = wcgain(ClosedLoop)
```

```
wcg = struct with fields:
           LowerBound: 1.5703e+00
           UpperBound: 1.5731e+00
    CriticalFrequency: 5.9267e+00
```

The worst-case gain is about 1.6. This analysis shows that while the controller K meets the disturbance rejection and noise insensitivity goals for the nominal plant, it is unable to maintain this level of performance for the specified level of plant uncertainty.

See Also

mussv | robgain | robstab | wcgain

Related Examples

- “Getting Reliable Estimates of Robustness Margins” on page 2-34

More About

- “Robustness and Worst-Case Analysis” on page 2-21

Getting Reliable Estimates of Robustness Margins

This example illustrates the pitfalls of using frequency gridding to compute robustness margins for systems with only real uncertain parameters. It presents a safer approach along with ways to mitigate discontinuities in the structured singular value μ.

How Discontinuities Can Hide Robustness Issues

Consider a spring-mass-damper system with 100% parameter uncertainty in the damping coefficient and 0% uncertainty in the spring coefficient. Note that all uncertainty is of `ureal` type.

```
m = 1;
k = 1;
c = ureal('c',1,'plusminus',1);
sys = tf(1,[m c k]);
```

As the uncertain element `c` varies, the only place where the poles can migrate from stable to unstable is at s = j*1 (1 rad/sec). No amount of variation in `c` can cause them to migrate across the jw-axis at any other frequency. As a result, the robust stability margin is infinite at all frequencies except 1 rad/s, where the margin with respect to the specified uncertainty is 1. In other words, the robust stability margin and the underlying structured singular value μ are discontinuous as a function of frequency.

The traditional approach to computing the robust stability margin is to pick a frequency grid and compute lower and upper bounds for μ at each frequency point. Under most conditions, the robust stability margin is continuous with respect to frequency and this approach gives good estimates provided you use a sufficiently dense frequency grid. However in problems with only `ureal` uncertainty, such as the example above, poles can migrate from stable to unstable only at specific frequencies (the points of discontinuity for μ), so any frequency grid that excludes these particular frequencies will lead to over-optimistic stability margins.

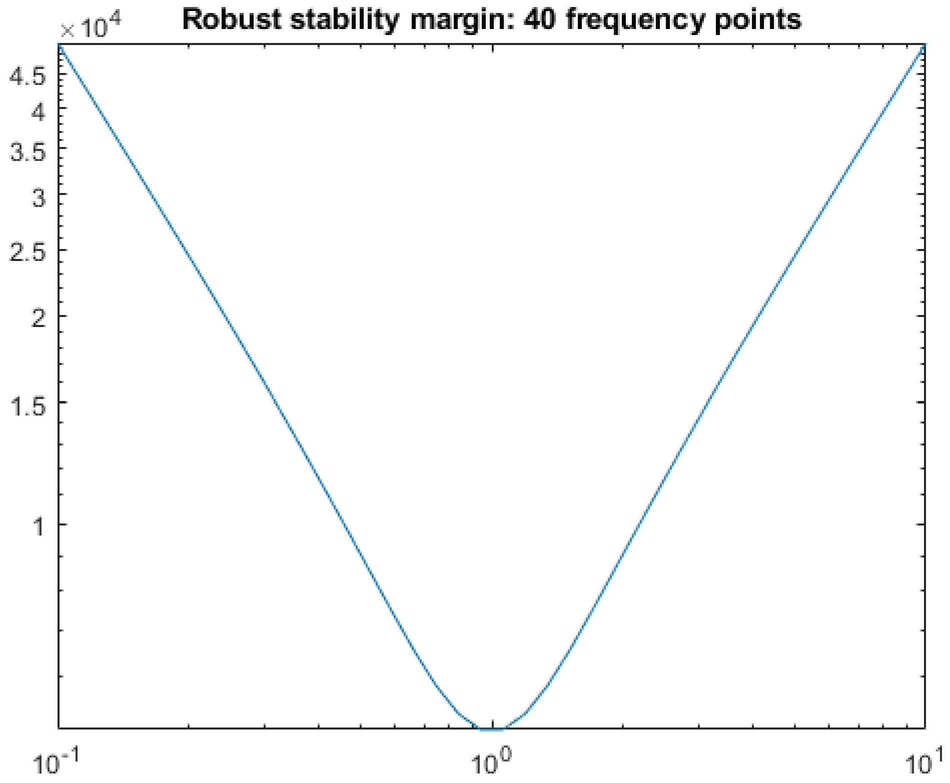
To see this effect, pick a frequency grid for the spring-mass-damper system above and compute the robust stability margins at these frequency points using `robstab`.

```
omega = logspace(-1,1,40); % one possible grid
[stabmarg,wcu,info] = robstab(sys,omega);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 5.0348e+03
           UpperBound: Inf
    CriticalFrequency: 0.1000
```

The field `info.Bounds` gives the margin lower and upper bounds at each frequency. Verify that the lower bound (the guaranteed margin) is large at all frequencies.

```
loglog(omega,info.Bounds(:,1))
title('Robust stability margin: 40 frequency points')
```



Note that making the grid denser would not help. Only by adding f=1 to the grid will we find the true margin.

```
f = 1;
stabmarg = robstab(sys,f)
```

```
stabmarg = struct with fields:
           LowerBound: 1.0000
           UpperBound: 1
    CriticalFrequency: 1
```

Safe Computation of Robustness Margins

Rather than specifying a frequency grid, apply `robstab` directly to the USS model `sys`. This uses a more advanced algorithm that is guaranteed to find the peak of μ even in the presence of a discontinuity. This approach is more accurate and often faster than frequency gridding.

```
[stabmarg,wcu] = robstab(sys)
```

```
stabmarg = struct with fields:
           LowerBound: 1.0000
           UpperBound: 1.0000
    CriticalFrequency: 1.0000
```

```
wcu = struct with fields:
    c: 2.2204e-16
```

This computes the correct robust stability margin (1), identifies the critical frequency (`f=1`), and finds the smallest destabilizing perturbation (setting `c=0`, as expected).

Modifying the Uncertainty Model to Eliminate Discontinuities

The example above shows that the robust stability margin can be a discontinuous function of frequency. In other words, it can have jumps. We can eliminate such jumps by adding a small amount of uncertain dynamics to every uncertain real parameter. This amounts to adding some dynamics to pure gains. Importantly, as the size of the added dynamics goes to zero, the estimated margin for the modified problem converges to the true margin for the original problem.

In the spring-mass-damper example, we model `c` as a `ureal` with the range [0.05,1.95] rather than [0,2], and add a `ultidyn` perturbation with gain bounded by 0.05. This combination covers the original uncertainty in `c` and introduces only 5% conservatism.

```
cc = ureal('cReal',1,'plusminus',0.95) + ultidyn('cUlti',[1 1],'Bound',0.05);
sysreg = usubs(sys,'c',cc);
```

Recompute the robust stability margin over the frequency grid `omega`.

```
[stabmarg,~,info] = robstab(sysreg,omega);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 2.3624
           UpperBound: 2.3630
    CriticalFrequency: 0.9427
```

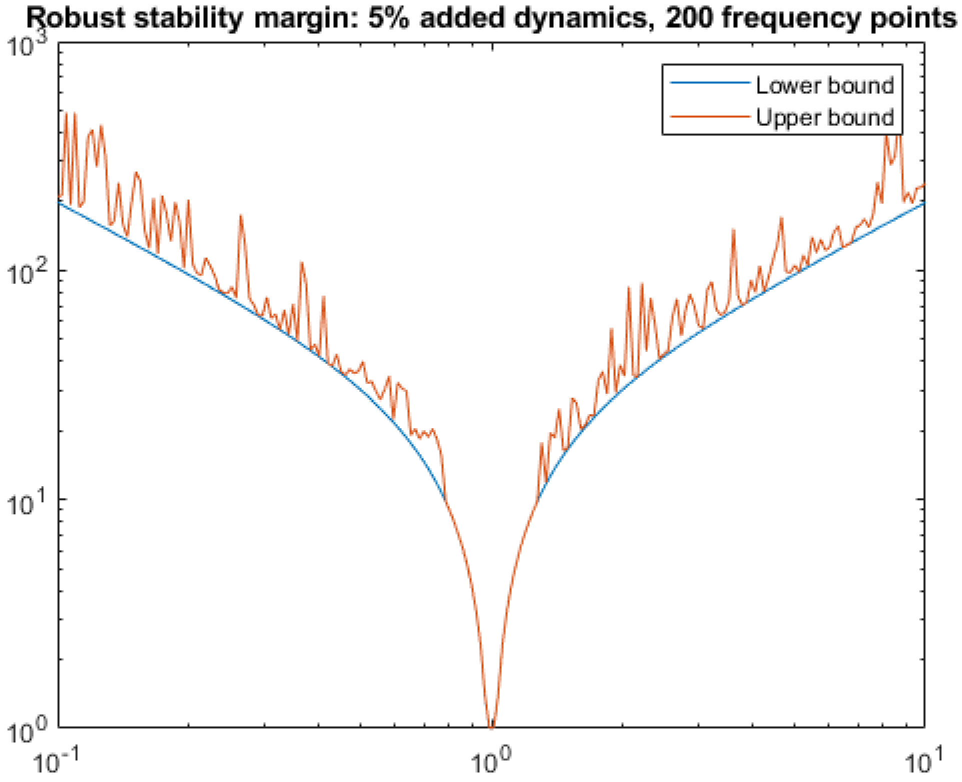
Now the frequency-gridded calculation yields a margin of 2.36. This is still greater than 1 (the true margin) because the density of frequency points is not high enough. Increase the number of points from 40 to 200 and recompute the margin.

```
OmegaDense = logspace(-1,1,200);
[stabmarg,~,info] = robstab(sysreg,OmegaDense);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 1.0026
           UpperBound: 1.0056
    CriticalFrequency: 0.9885
```

Plot the robustness margin as a function of frequency.

```
loglog(OmegaDense,info.Bounds(:,1),OmegaDense,info.Bounds(:,2))
title('Robust stability margin: 5% added dynamics, 200 frequency points')
legend('Lower bound','Upper bound')
```



The computed margin is now close to 1, the true margin for the original problem. In general, the stability margin of the modified problem is less than or equal to that of the original problem. If it is significantly less, then the answer to the question "What is the stability margin?" is very sensitive to the uncertainty model. In this case, we put more faith in the value that allows for a few percents of unmodeled dynamics. Either way, the stability margin for the modified problem is more trustworthy.

Automated Regularization of Discontinuous Problems

The command `complexify` automates the procedure of replacing a `ureal` with the sum of a `ureal` and `ultidyn`. The analysis above can be repeated using `complexify` obtaining the same results.

```
sysreg = complexify(sys,0.05,'ultidyn');
[stabmarg,~,info] = robstab(sysreg,OmegaDense);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 1.0026
           UpperBound: 1.0056
    CriticalFrequency: 0.9885
```

Note that such regularization is only needed when using frequency gridding. Applying `robstab` directly to the original uncertain model `sys` yields the correct margin without frequency gridding or need for regularization.

References

The continuity of the robust stability margin, and the subsequent computational and interpretational difficulties raised by the presence of discontinuities are considered in [1]. The consequences and interpretations of the regularization illustrated in this small example are described in [2]. An extensive analysis of regularization for 2-parameter example is given in [2].

[1] Barmish, B.R., Khargonekar, P.P, Shi, Z.C., and R. Tempo, "Robustness margin need not be a continuous function of the problem data," Systems & Control Letters, Vol. 15, No. 2, 1990, pp. 91-98.

[2] Packard, A., and P. Pandey, "Continuity properties of the real/complex structured singular value," Vol. 38, No. 3, 1993, pp. 415-428.

See Also

`diskmargin` | `robstab` | `wcdiskmargin`

Related Examples

- “Robust Stability, Robust Performance and Mu Analysis” on page 2-27

More About

- “Robustness and Worst-Case Analysis” on page 2-21

Mu Synthesis

Robust Controller Design Using Mu Synthesis

The technique of μ synthesis extends the methods of H_∞ synthesis to design a robust controller for an uncertain plant. You can perform μ synthesis on plants with parameter uncertainty, dynamic uncertainty, or both using the `musyn` command.

`musyn` seeks a controller that minimizes the robust H_∞ performance of the closed-loop system. The robust H_∞ performance, also called μ, quantifies how modeled uncertainty affects the performance of a feedback loop. For details about μ and how it is computed, see “Robust Performance Measure for Mu Synthesis” on page 3-11.

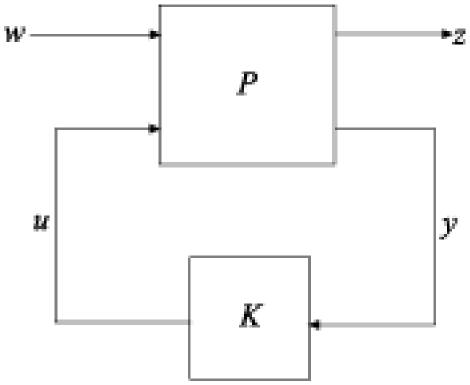
Basic μ Synthesis Workflow

You can use `musyn` to:

- Synthesize "black box" unstructured robust controllers.
- Robustly tune a fixed-order or fixed-structure controller made up of tunable components such as PID controllers, state-space models, and static gains.

μ Synthesis of Unstructured Controllers

μ synthesis of unstructured controllers is analogous to controller synthesis with `hinfsyn`, except that the plant includes uncertainty. As with `hinfsyn`, you set up your problem as the feedback system `CL = lft(P,K)`, where `P` is the plant and `K` is the controller to design.



In the diagram:

- *w* represents the disturbance inputs.
- *u* represents the control inputs.
- *z* represents the error outputs to be kept small.
- *y* represents the measurement outputs provided to the controller.

You construct the uncertain plant *P* by building a state-space model with uncertain coefficients (`ureal` or `ucomplex`) blocks, uncertain dynamics (`ultidyn` blocks), or both. Construct the plant such that the measurement outputs *y* are the last outputs, and the control inputs *u* are the last inputs. As with `hinfsyn`, you can optionally augment the plant inputs and outputs with weighting functions (loop-shaping filters) that represent control objectives.

You then pass this plant to musyn, which seeks a controller K that minimizes the robust H_∞ performance on page 3-11. The controller is returned as a state-space model. For a simple example, see “Unstructured Robust Controller Synthesis” on the musyn reference page.

μ Synthesis of Fixed-Structure Controllers

Instead of obtaining a controller that is a free-form state-space model, you can specify a fixed controller structure with tunable parameters. musyn then adjusts those parameters to minimize the robust H_∞ performance of the system. μ synthesis of fixed-structure controllers is analogous to controller tuning with hinfstruct, except that the plant includes uncertainty.

To set up your problem for fixed-structure μ synthesis, you construct a generalized state-space (genss) model of the uncertain closed-loop system with tunable controller elements. To do so, you create and interconnect:

- Numeric LTI models representing the fixed components of the control system
- Uncertain control design blocks such as ureal and ultidyn blocks representing the uncertain components of the plant
- Optional LTI weighting functions (loop-shaping filters) that represent control objectives
- Tunable control design blocks such as tunablePID, tunableSS, and tunableGain to represent tunable components of the system

For an example that shows how to build such a model, see “Build Tunable Control System Model With Uncertain Parameters”.

You pass the tunable, uncertain closed-loop model to musyn, which seeks values of the tunable parameters that optimize the robust H_∞ performance from the model inputs to its outputs. For a simple example, see “Robust Tuning of Fixed-Structure Controller” on the musyn reference page.

If you have a Simulink® model of your control system, you can use slTuner to linearize the model with specified uncertain parameters and tunable blocks. You then use getIOTransfer to extract a genss model for controller design with musyn. For an example, see “Model Uncertainty in Simulink for Robust Tuning”.

Interpret the Results of μ Synthesis

musyn returns a robust controller K (for unstructured controller tuning) or a tuned version of the control system CL (for fixed-structure controller tuning). It also returns the best achieved robust H_∞ performance as the CLperf output argument. This value tells you that with the controller returned by musyn, the peak gain of the closed-loop system remains below CLperf for uncertainty up to 1/CLperf in normalized units. For example:

- CLperf = 0.5 means that the closed-loop gain remains below 0.5 for uncertainty up to twice the uncertainty specified in the input model. The worst-case gain for the specified uncertainty is typically smaller.
- CLperf = 2 means that the closed-loop gain remains below 2 for uncertainty up to half the uncertainty specified in *CL*. For this value, the worst-case gain for the full specified uncertainty can be much larger. It can even be infinite, meaning that the system does not remain stable over the full range of the specified uncertainty.

For more detailed information about this quantity and how it is computed, see “Robust Performance Measure for Mu Synthesis” on page 3-11.

To find K, musyn uses an iterative process called D-K iteration. This process solves a sequence of scaled H_∞ problems. The frequency-dependent scalings, called *D* and *G* scalings, take advantage of the uncertainty structure. To perform D-K iteration, musyn:

1. Uses H_∞ synthesis to find a controller that minimizes the closed-loop gain of the nominal system.
2. Performs a robustness analysis to estimate the robust H_∞ performance of the closed-loop system. This quantity is expressed as a scaled H_∞ norm involving the *D* and *G* scalings (the *D* step).
3. Finds a new controller to minimize the H_∞ norm obtained in step 2 (the *K* step).
4. Repeats steps 2 and 3 until the robust performance stops improving.

For mathematical details about how this algorithm works, see “D-K Iteration Process” on page 3-16.

musyn gives you two ways to monitor and interpret the progress of the algorithm: the default display and the full display.

Default musyn Display

By default, musyn provides a brief display of algorithm progress in the MATLAB command window. For instance:

```
DG-K ITERATION SUMMARY:
-------------------------------------------------------------------
                       Robust performance                 Fit order
-------------------------------------------------------------------
  Iter         K Step       Peak MU       DG Fit           D      G
    1             100        5.747        6.394           10      4
    2           5.221        3.433        4.607           10      6
    3           2.682        2.263        2.627           10      4
    4           1.987        1.687         2.18           10      6
    5           1.287        1.192        1.377           10      8
    6           1.079        1.087         1.09           10      8
    7           1.076        1.046        1.055            8      6
    8           1.049        1.024        1.044           10      6
    9           1.045        1.022        1.039            8      6
   10            1.04        1.023        1.033            8      6

Best achieved robust performance: 1.02
```

The display includes information about each D-K iteration.

- K Step column — For the first iteration, this value is the H_∞ performance of the closed-loop nominal system after controller synthesis. For remaining iterations, this column shows the scaled H_∞ norm after controller synthesis.
- Peak MU column — Robust performance ($\bar{\mu}$, an upper bound on μ) for the controller designed in the K Step.
- DG fit column — Scaled H_∞ performance after fitting the *D* and *G* scalings with rational functions.
- Fit order columns — Orders of the rational function used to fit the scalings in that iteration. If the system has only complex uncertainty, or when the 'MixedMU' option of musynOptions is set to 'off', then musyn does not apply *G* scaling. In that case, only the *D* fit order is listed.

If you see a large difference between the `Peak MU` and `DG Fit` values in a given iteration, it is a sign that `musyn` cannot find a good fit for the scalings. In that case, you can try increasing the maximum fit order using the `'FitOrder'` option of `musynOptions`.

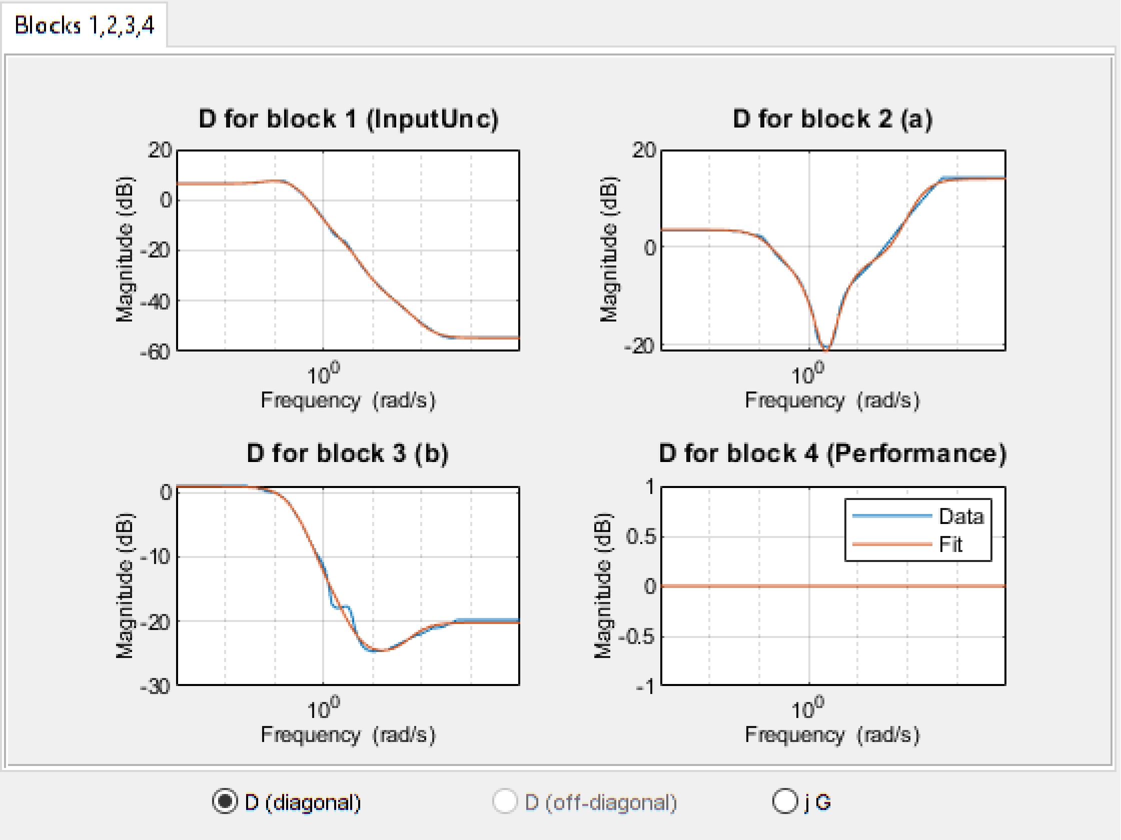
For other ways to improve the results, see “Improve Results of Mu Synthesis” on page 3-8.

Full musyn Display

You can obtain a more detailed view into the progress of D-K iteration by setting the `'Display'` option of `musynOptions` to `'full'`. If you turn on the full display, then `musyn` pauses after each D-K iteration so that you can view the detailed results of the iteration. In addition to the information described in “Default musyn Display” on page 3-4, the full display:

- Shows detailed computation information for the controller synthesis (*K* step) of the current iteration. For unstructured controllers, see `hinfsyn` for information about this display. For fixed-structure controllers, see `hinfstructOptions`.
- Shows information about the fits for the *D* scalings and the *G* scalings (if any) of the current iteration. The information includes the fit order of the scalings for each uncertain block. It also includes a goodness-of-fit score. A score less than or equal to 1 indicates adequate fit for μ synthesis.
- Generates plots that let you visualize the *D* and *G* fits, the robust performance before fitting, and the scaled H_∞ performance after fitting. Examining these plots can help you determine if the maximum fit order is high enough to capture all the frequency-dependent variation in the scalings (see the `FitOrder` option of `musynOptions` for more information).

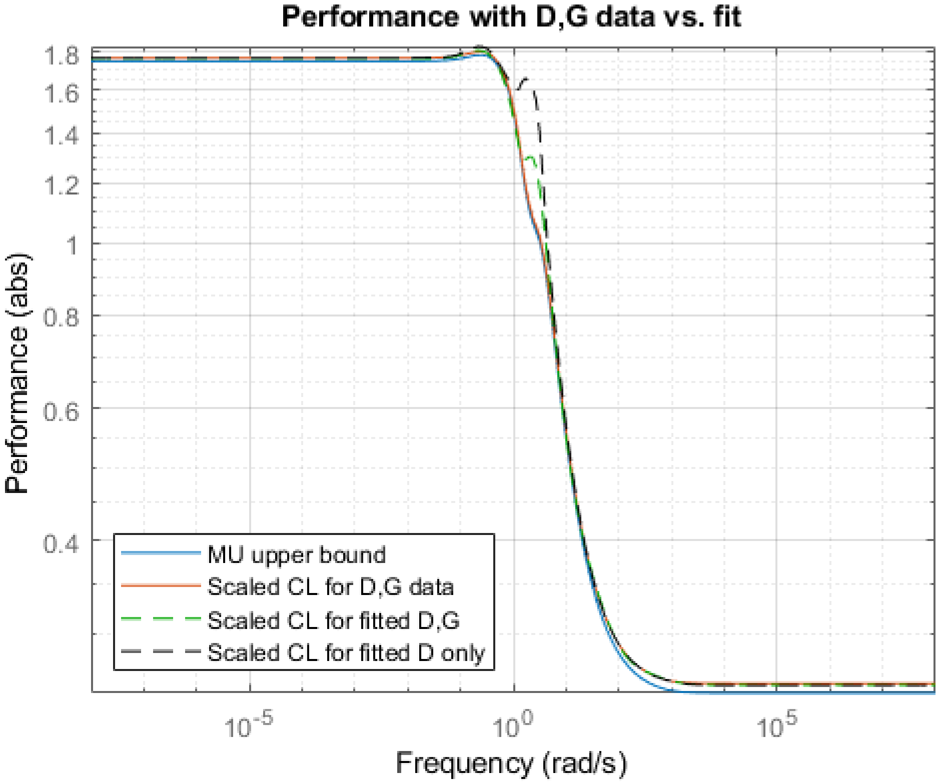
The `D Fit` or `D,G Fit` plot shows the scaling data and the corresponding rational fits.



Use the radio buttons to select which scalings to inspect:

- `D (diagonal)` shows the magnitude of the diagonal elements of the *D* scalings.
- `D (offdiagonal)` shows the magnitude and phase of the off-diagonal elements of the *D* scalings. This plot is available when your system has repeated uncertain blocks. (See the `'FullDG'` option of `musynOptions` for more information.)
- `jG` shows the magnitude and phase of the *G*-scalings. *G* scalings are present only when there is real uncertainty and the `MixedMU` option of `musynOptions` is `'on'`. See “Improve Results of Mu Synthesis” on page 3-8.

The `Robust Performance` plot shows the performance of the closed-loop system before and after fitting.



The traces on this plot are:

- `Mu upper bound` — Robust performance, the upper bound $\bar{\mu}$ as a function of frequency
- `Scaled CL for D,G data` — Scaled H_∞ performance before fitting the D and G scaling data with rational functions
- `Scaled CL for fitted D,G` — Scaled H_∞ performance after fitting
- `Scaled CL for fitted D only` — Shows what behavior would not be captured if G were omitted

For detailed information about the D-K iteration algorithm and the meaning of all the quantities in the full display, see “D-K Iteration Process” on page 3-16.

See Also

`musyn` | `musynOptions`

More About

- “Improve Results of Mu Synthesis” on page 3-8
- “Robust Performance Measure for Mu Synthesis” on page 3-11
- “First-Cut Robust Design” on page 3-20
- “Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis” on page 3-77

Improve Results of Mu Synthesis

You can sometimes improve the results of robust controller synthesis with musyn. Even if the default options yield good results, by changing certain options, you might be able to:

- Find a controller that yields better robust performance.
- Find a lower order controller that yields similar robust performance.

Consider trying several of the techniques described here to see whether the results you obtain from musyn can be improved.

Mixed-µ Synthesis for Real Uncertainty

By default, musyn treats all uncertainties as complex uncertainties, even those represented by real parameters. For ureal blocks, musyn assumes that each real parameter has an imaginary part that can vary by the same amount as the real part. This assumption simplifies the computation, but yields a more conservative estimate of the robust performance of the system.

When you have real uncertainty, you can instead use mixed-*μ* synthesis, which explicitly takes into account the fact that some uncertain parameters are limited to real values. Try using mixed-*μ* synthesis to see if it improves the performance relative to the controller you obtain without it.

To use mixed-*μ* synthesis, set the 'MixedMU' option of musynOptions to 'on'. For an example that illustrates the benefit of mixed-*μ* synthesis, see “Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis” on page 3-77.

Mixed-*μ* synthesis complicates the computation and can result in higher order controllers. The techniques in “Reduce Controller Order” on page 3-8 can help simplify the resulting controller.

Reduce Controller Order

For unstructured controller design, musyn can return relatively high-order controllers. musyn uses frequency-dependent scaling matrices that are fit by rational functions. (See “D-K Iteration Process” on page 3-16.) The order required to fit the scalings and the number of uncertain blocks in your system contribute to the order of the final optimized controller. Therefore, after using musyn for an initial robust controller design, it can be useful to search for a lower order controller that achieves similar robust performance. Among approaches to obtaining a lower order controller, you can:

- Reduce the order of controller returned by musyn on page 3-8.
- Use a lower order fixed-structure controller on page 3-9.
- Reduce the maximum scaling order on page 3-9.
- Use diagonal scalings for repeated uncertain blocks on page 3-9.

Reduce Order of Returned Controller

One technique is to use model-reduction commands to reduce the controller that musyn returns, and find the lowest order approximation that achieves similar performance. For an example illustrating this approach, see the musynperf reference page.

Even if the initial controller you obtain with musyn is not reducible in a way that preserves robust performance, a lower order controller that achieves the same performance might exist. Consider trying the other techniques to see if varying parameters of the musyn computation can help you find such a controller.

Lower Order Fixed-Structure Controller

This approach takes advantage of the ability of musyn to tune fixed-structure controllers. Suppose that you use musyn to design a full-order, centralized controller K for an uncertain plant P with nmeas measurement signals and ncont control signals. You can create a fixed-order, tunable state-space model of a lower order than K, and use musyn again to tune the free parameters of that model. If the new controller achieves robust performance close to that of the unstructured controller, try again with an even lower order tunable state-space model. For instance, suppose K is a 10th-order controller returned by musyn for the plant P. The following commands create and tune a fifth-order state-space controller by forming the closed-loop uncertain system with the tunable controller and passing it to musyn.

```
C0 = tunableSS('C0',5,nmeas,ncont);
CL0 = lft(P,C0);
[CL,CLperf,info] = musyn(CL0);
```

For a simple example, see “Robust Tuning of Fixed-Structure Controller” on the musyn reference page.

Reduce Maximum Scaling Order

For each iteration, musyn fits each entry in the *D* and *G* scaling matrices by a rational function of automatically selected order. The higher the order of these functions, the higher the order of the resulting controller. By default, the maximum order is 5 for *D* scaling, and 2 for the *G* scaling matrices. If these defaults yield a controller with good robust performance, try lowering the maximum order to see if musyn returns a lower-order controller with similar performance. To change the maximum order, use the 'FitOrder' option of musynOptions.

Diagonal Scalings for Repeated Blocks

If your system has repeated uncertain parameters, you can restrict the *D* and *G* scalings so they are diagonal, which can result in a lower order unstructured controller. For more information, see “Repeated Parameter Blocks” on page 3-9.

Repeated Parameter Blocks

An uncertain parameter can occur multiple times in a given model. For example, the following code creates an uncertain state-space model that has two occurrences each of the uncertain parameters p1 and p2.

```
p1 = ureal('p1',10);
p2 = ureal('p2',3);
A = [-p1 p2;0 -p1];
B = [-p2; p2];
C = [1 0;1 1];
D = [0;0];
sys = ss(A,B,C,D)
```

```
sys =

  Uncertain continuous-time state-space model with 2 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p1: Uncertain real, nominal = 10, variability = [-1,1], 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-1,1], 2 occurrences
```

```
Type "sys.NominalValue" to see the nominal value, "get(sys)" to see all properties,
and "sys.Uncertainty" to interact with the uncertain elements.
```

Multiple occurrences of uncertain parameters can increase the order of the rational fit functions for the *D* and *G* scalings. Therefore, they can increase the number of states in the controller returned by `musyn`. You can mitigate this effect of repeated parameters in several ways:

- Use diagonal scalings on page 3-10.
- Reduce repetitions of uncertain parameters in the plant on page 3-10.
- Use `systune` on page 3-10.

Use Diagonal Scalings

By default, `musyn` by default uses full matrices for the *D* and *G* scalings of repeated blocks. Full scaling matrices can have frequency-dependent entries both on and off the diagonal. Fitting all of these entries can result in high controller order. If `musyn` instead uses diagonal scaling, then fewer independent fit functions are needed, which can reduce the controller order.

Diagonal scaling, is equivalent to treating each repeated block as an independent instance of the uncertain parameter. Therefore, full scaling is less conservative and can yield better robust performance. However, to reduce the controller order, you can try diagonal scaling and see whether `musyn` can still find an adequate controller using the more conservative estimation of the μ upper bound.

To specify diagonal scaling for repeated blocks, use the `'FullDG'` option of `musynOptions`.

Reduce Repetitions in the Plant

Use `simplify` to reduce the number of repeated parameters in the plant before calling `musyn`. The `simplify` command tries to remove redundant instances of uncertain blocks.

Use systune

If you have more than about five repeated instances of an uncertain parameter and have no dynamic uncertainty (no `ultidyn` blocks), consider using `systune` instead of `musyn`. The `systune` command tunes fixed-structure controller elements. It can perform robust controller tuning without degradation caused by large numbers of repeated blocks. For more information on ways to perform robust tuning with `systune`, see “Robust Tuning Approaches”.

See Also

`musyn` | `musynOptions`

More About

- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “Robust Performance Measure for Mu Synthesis” on page 3-11
- “Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis” on page 3-77
- “Simultaneous Stabilization Using Robust Control” on page 3-54

Robust Performance Measure for Mu Synthesis

The robust H_∞ performance quantifies how modeled uncertainty affects the performance of a feedback loop. Performance here is measured with the H_∞ norm (peak gain) of a transfer function of interest, such as that from disturbance to error signals. (See "H-Infinity Performance".)

For a system $T(s)$, the robust H_∞ performance μ is the smallest value γ such that the peak gain of T remains below γ for uncertainty up to $1/\gamma$, in normalized units. For example:

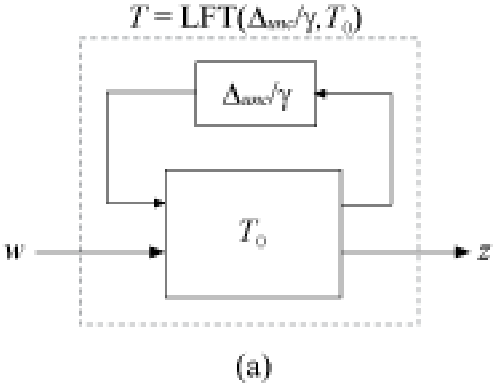
- $\mu = 0.5$ means that $||T(s)||_\infty$ remains below 0.5 for uncertainty up to twice the uncertainty specified in T. The worst-case gain for the specified uncertainty is typically smaller.
- $\mu = 2$ means that $||T(s)||_\infty$ remains below 2 for uncertainty up to half the uncertainty specified in T. For this value, the worst-case gain for the full specified uncertainty can be much larger. It can even be infinite, meaning that the system does not remain stable over the full range of the specified uncertainty.

The quantity μ is the peak value over frequency of the structured singular value $\mu(\omega)$ for the uncertainty specified in T. This quantity is a generalization of the singular value for uncertain systems. It depends on the structure of the uncertainty in the system. In practice, μ is difficult to compute exactly, so the software instead computes lower and upper bounds, $\underline{\mu}$ and $\bar{\mu}$. The upper bound $\bar{\mu}$ has several applications in control system design and analysis. You can:

- Use `musyn` to design a controller for an uncertain plant that minimizes $\bar{\mu}$ of the closed-loop system. In addition to the resulting controller, `musyn` returns the corresponding value of $\bar{\mu}$ in the `CLperf` output argument.
- Use `musynperf` evaluate the robust performance of an uncertain system. This function returns lower and upper bounds on μ, the uncertainty values that yield the peak μ, and other information about the closed-loop robust performance.

Uncertain Model

To understand the computation of robust H_∞ performance, consider an uncertain system $T(s)$, modeled as a fixed portion T_0 and an uncertain portion Δ_{unc}/γ.



Δ_{unc} collects the uncertain elements $\{\Delta_1,\ldots,\Delta_N\}$.

$$\Delta_{unc} = \begin{pmatrix} \Delta_1 & & \\ & \ddots & \\ & & \Delta_N \end{pmatrix}.$$

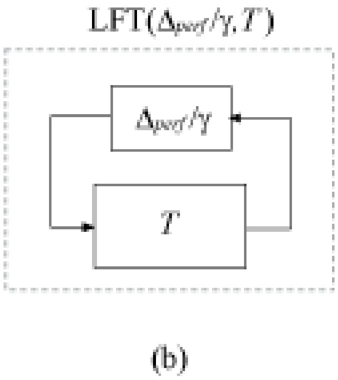
Each Δ_j is an arbitrary real, complex, or dynamic uncertainty that is normalized such that $||\Delta_j||_\infty \leq 1$. The factor γ adjusts the level of uncertainty.

Robust Performance as a Robust Stability Margin

Suppose that for the system modeled as in diagram (a),

$||T||_\infty \leq \gamma$ for all $||\Delta_{unc}||_\infty \leq 1$.

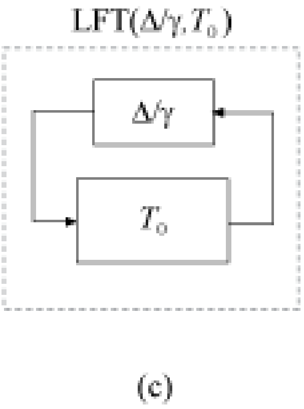
By the small-gain theorem (see [1]), this robust performance condition is equivalent to stating that the system of diagram (b), LFT(Δ_{perf}/γ,T), is stable for all for all $||\Delta_{perf}||_\infty \leq 1$.



Δ_{perf} is called the performance block. Expand T as in diagram (a), and group Δ_{perf} with the uncertain blocks Δ_{unc} to define a new block Δ,

$$\Delta \triangleq \begin{pmatrix} \Delta_{perf} & 0 \\ 0 & \Delta_{unc} \end{pmatrix}.$$

The result is the system in the following diagram.



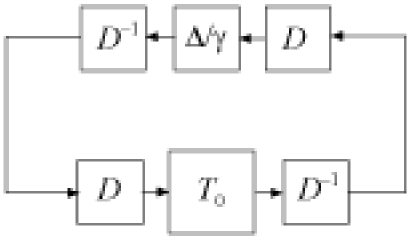
Thus, the robust performance condition on the system of diagram (a) is equivalent to a stability condition on diagram (c), or

$$\|T\|_\infty \leq \gamma \text{ for all } \|\Delta_{unc}\|_\infty \leq 1 \quad \Leftrightarrow \quad \|LFT(\Delta/\gamma), T_0\|_\infty \text{ stable for all } \|\Delta\|_\infty \leq 1.$$

The robust performance μ is the smallest γ for which this stability condition holds. Equivalently, $1/\mu$ is the largest uncertainty level $1/\gamma$ for which the system of diagram (c) is robustly stable. In other words, $1/\mu$ is the robust stability margin of the feedback loop of diagram (c) for the augmented uncertainty Δ. (For more information on robust stability margins, see "Robustness and Worst-Case Analysis" on page 2-21.)

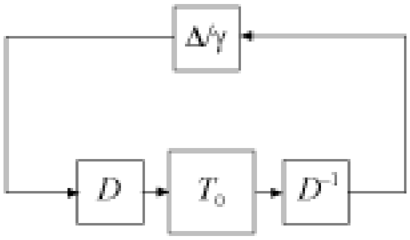
Upper Bound of μ

To obtain an estimate on the upper bound of μ, the software introduces scalings. If the system in diagram (c) is stable for all $||\Delta||_\infty \leq 1$, then the system of the following diagram is also stable, for any invertible D.



(d)

If D commutes with Δ, then the system of diagram (d) is the same as the system in the following diagram.



(e)

The matrices D that structurally commute with Δ are called D scalings. They can be frequency dependent, which is denoted by $D(\omega)$.

Define $\bar{\mu}$ as:

$$\bar{\mu} \triangleq \inf_{D(\omega)} \left\| D(\omega) T_0(j\omega) D(\omega)^{-1} \right\|_\infty .$$

For the optimal $D^*(\omega)$, and any $\gamma \geq \bar{\mu}$,

$$\left\| D^*(\omega) T_0(j\omega) D^*(\omega)^{-1} \right\|_\infty \leq \gamma .$$

Therefore, by the small-gain theorem, the system of diagram (e) is stable for all $||\Delta||_\infty \leq 1$. It follows that $1/\gamma \leq 1/\mu$, or $\gamma \leq \mu$, because $1/\mu$ is the robust stability margin. Consequently, $\mu \leq \bar{\mu}$, so that $\bar{\mu}$ is an upper bound for the robust performance μ. This upper bound $\bar{\mu}$ is the quantity computed by `musynperf` and optimized by `musyn`.

D and G Scalings

When all the uncertain elements Δ_j are complex or LTI dynamics, the software approximates $\bar{\mu}$ by picking a frequency grid $\{\omega_1,\ldots,\omega_N\}$. At each frequency point, the software solves the optimal scaling problem

$$\bar{\mu}_i = \inf_{D_i} \| D_i T_0(j\omega_i) D_i{}^{-1} \| .$$

It then sets $\bar{\mu}$ to the largest result over all frequencies in the grid,

$$\bar{\mu} = \max_i \bar{\mu}_i .$$

When some Δ_j are real, it is possible to obtain a less conservative upper bound by using additional scalings called G scalings. In this case, $\bar{\mu}$ is the smallest $\bar{\mu}_i$ over frequency such that

$$\begin{pmatrix} T_0(j\omega_i) \\ I \end{pmatrix}^H \begin{pmatrix} D_r(\omega_i) & -jG_{cr}^H(\omega_i) \\ jG_{cr}(\omega_i) & -\bar{\mu}_i^2 D_c(\omega_i) \end{pmatrix} \begin{pmatrix} T_0(j\omega_i) \\ I \end{pmatrix} \leq 0$$

for some $D_r(\omega_i)$, $D_c(\omega_i)$, and $G_{cr}(\omega_i)$. These frequency-dependent matrices are the D and G scalings.

Mu Synthesis

The `musyn` command synthesizes robust controllers using an iterative process that optimizes the robust performance $\bar{\mu}$. To learn how to use `musyn`, see “Robust Controller Design Using Mu Synthesis” on page 3-2. For details about the `musyn` algorithm, see “D-K Iteration Process” on page 3-16.

References

[1] Skogestad, S. and I. Postlethwaite, *Multivariable Feedback Control: Analysis and Design*, 2d ed. West Sussex, England: John Wiley & Sons, 2005, pp. 156, 306.

See Also

`musyn` | `musynperf`

More About

- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “D-K Iteration Process” on page 3-16

D-K Iteration Process

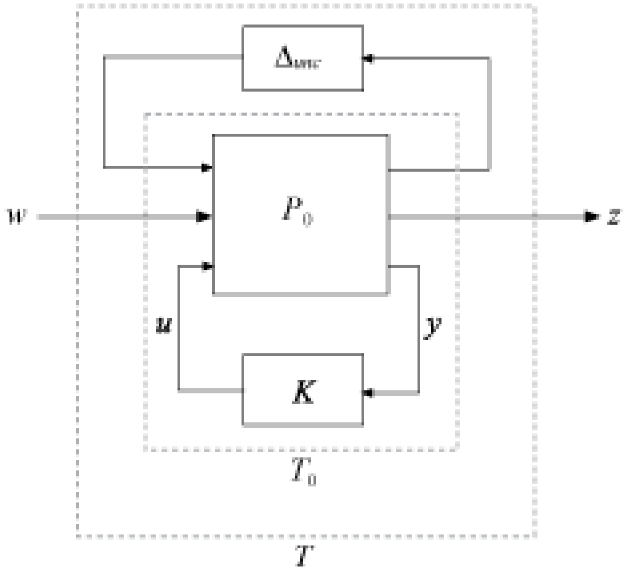
You can use the `musyn` command to design a robust controller for an uncertain plant, as described in "Robust Controller Design Using Mu Synthesis" on page 3-2. The algorithm used by `musyn` is an iterative process called D-K iteration. In this process, the function:

1. Uses H_∞ synthesis to find a controller that minimizes the closed-loop gain of the nominal system.
2. Performs a robustness analysis to estimate the robust H_∞ performance of the closed-loop system. This quantity is expressed as a scaled H_∞ norm involving dynamic scalings called the *D* and *G* scalings (the *D* step).
3. Finds a new controller to minimize the scaled H_∞ norm obtained in step 2 (the *K* step).
4. Repeats steps 2 and 3 until the robust performance stops improving.

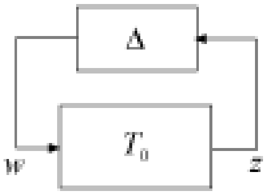
Both the *D* step and *K* step are mathematically intensive computations. Details of the algorithm follow.

D Step

In the *D* step, `musyn` computes the upper bound $\bar{\mu}$ of the robust H_∞ performance for the current controller *K*. The *D* step begins with a robust performance analysis for the closed-loop uncertain system *T* = LFT(*P*,*K*), as in the following diagram.



Introducing a performance block Δ_{perf} transforms the robust-performance analysis of *T* to a robust-stability analysis of the feedback loop in the following diagram.



Here, Δ is the augmented uncertainty structure

$$\Delta \triangleq \begin{pmatrix} \Delta_{perf} & 0 \\ 0 & \Delta_{unc} \end{pmatrix}.$$

`musyn` computes $\bar{\mu}$, an upper bound on the robust H_∞ performance. To do so, `musyn` selects a frequency grid $\{\omega_1,\ldots,\omega_N\}$. For T with complex uncertainty only, `musyn` computes at each frequency ω_i

$$\bar{\mu}_i = \inf_{D_i} \| D_i T_0(j\omega_i) D_i{}^{-1} \| .$$

The frequency-dependent matrices D, which commute with Δ, are called D scalings. $\bar{\mu}$ is the largest result over all frequencies in the grid,

$$\bar{\mu} = \max_i \bar{\mu}_i .$$

When you use `musyn`, you can access the results of the D step in several ways.

- The default `musyn` display shows $\bar{\mu}$ for each iteration in the `Peak MU` column.
- `musyn` returns $\bar{\mu}$ for each iteration in the `PeakMU` field of the `info` output argument.
- `musyn` returns D_i in the `DG` field of the `info` output argument.
- To visualize the frequency-dependence of D_i, set the `'Display'` option of `musynOptions` to `'full'`.

For additional details about the computation and interpretation of $\bar{\mu}$, see “Robust Performance Measure for Mu Synthesis” on page 3-11.

D-Fitting and Scaled H∞ Performance

`musyn` fits a rational function $D(s)$ to the sequence of scalings $\{D_i\}$. The fit yields a quantity μ_F called the scaled H_∞ performance,

$$\mu_F \triangleq \left\| D(T_0)D^{-1} \right\|_\infty .$$

Because the fit is not exact, μ_F is typically somewhat larger than $\bar{\mu}$.

You can access the results of the fit in several ways.

- The default `musyn` display shows μ_F for each iteration in the `DG Fit` column.
- `musyn` returns μ_F for each iteration in the `PeakMUFit` field of the `info` output argument.
- `musyn` returns the fitting functions in the `dr` and `dc` fields of the `info` output argument.
- To visualize the frequency dependence of the fitting functions, set the `'Display'` option of `musynOptions` to `'full'`.

K Step

T_0 depends on the choice of controller K by the relation T_0 = LFT(P_0,K). Therefore, minimizing μ_F with respect to K is a scaled H_∞ synthesis problem. Thus, in the K step, `musyn` uses `hinfsyn` or `hinfstruct` to compute a controller K^* that minimizes μ_F. The minimized quantity is the scaled H_∞ norm. For the algorithm to make progress, the new controller must improve the robust performance obtained in the D step:

$$\left\| D\,\mathrm{LFT}(P_0, K^*)\,D^{-1} \right\|_\infty < \bar{\mu}\,.$$

Otherwise, the progress is not sufficient for compensate for fitting errors. Thus `musyn` terminates D-K iteration process when K^* does not improve the robust performance within the tolerance specified by the `'TolPerf'` option of `musynOptions`.

You can access the results of the K step in several ways.

- The default `musyn` display shows the scaled H_∞ norm for each iteration in the `K Step` column.
- `musyn` returns the new controller in the `K` of the `info` output argument, and the corresponding scaled H_∞ norm for each iteration in the `gamma` field.

Mixed Real and Complex Uncertainty

When the system has both real and complex uncertainty and you set the `'MixedMU'` option of `musynOptions` to `'on'`, `musyn` uses an additional G-scaling to improve the computation of $\bar{\mu}$. The algorithm in this case is called mixed-µ synthesis.

For mixed uncertainty, `musyn` computes $\bar{\mu}_i$ and scalings $D_r(\omega_i)$, $D_c(\omega_i)$, and $G_{cr}(\omega_i)$ such that

$$\begin{pmatrix} T_0(j\omega_i) \\ I \end{pmatrix}^H \begin{pmatrix} D_r(\omega_i) & -jG_{cr}^H(\omega_i) \\ jG_{cr}(\omega_i) & -\bar{\mu}_i^2 D_c(\omega_i) \end{pmatrix} \begin{pmatrix} T_0(j\omega_i) \\ I \end{pmatrix} \le 0$$

at each frequency in the grid.

`musyn` fits the D and G scaling data by constructing a rational function

$$F(s) = \Psi(s)\begin{pmatrix} d_r(s) & 0 \\ 0 & d_c(s) \end{pmatrix}$$

such that

- $d_r(s)$, $d_c(s)$, and $\Psi(s)$ are stable with stable inverse.
- $d_r(s)$ and $d_c(s)$ approximate the square roots of the diagonal entries of $D_r(\omega_i)$ and $D_c(\omega_i)$.
- F approximately satisfies

$$\begin{pmatrix} D_r(\omega_i) & -jG_{cr}^H(\omega_i) \\ jG_{cr}(\omega_i) & -\mu^2 D_c(\omega_i) \end{pmatrix} \approx F(j\omega_i)^H J\, F(j\omega_i),$$

$$J = \begin{pmatrix} I_r & 0 \\ 0 & -I_c \end{pmatrix}.$$

Finally, the scaled H_∞ performance is defined as

$$\mu_F \triangleq \|\bar{T}(s)\|_\infty,$$

where $\bar{T}(s)$ is the transformed system,

$$\bar{T}(s) \triangleq \bar{\mu}\bar{T}_1\bar{T}_2^{-1}, \quad \begin{pmatrix} \bar{T}_1(s) \\ \bar{T}_2(s) \end{pmatrix} \triangleq \Psi(s)\begin{pmatrix} d_r(s)T_0(s) \\ d_c(s) \end{pmatrix}.$$

For an exact fit of D and G, $\|\bar{T}(j\omega_i)\| = \bar{\mu}_i$. Therefore, in general, $\mu_F \geq \bar{\mu}$.

Because the transformed system $\bar{T}(s)$ is still a linear fractional function of the controller K, the K step for the mixed-μ case proceeds by computing a controller K* that minimizes $\|\bar{T}\|_\infty$.

When using `musyn`, you can access the D and G scalings in several ways.

- `musyn` returns the D and G scaling data in the `DG` field of the `info` output argument.
- `musyn` returns the fitting functions in the `dr`, `dc`, and `PSI` fields of the `info` output argument.
- To visualize the frequency dependence of the scaling data and fitting functions, set the `'Display'` option of `musynOptions` to `'full'`.

See Also

`musyn` | `musynOptions` | `musynperf`

More About

- “Robust Performance Measure for Mu Synthesis” on page 3-11
- “Robust Controller Design Using Mu Synthesis” on page 3-2

First-Cut Robust Design

This example shows how to use the Robust Control Toolbox™ commands `usample`, `ucover` and `musyn` to design a robust controller with standard performance objectives. It can serve as a template for more complex robust control design tasks.

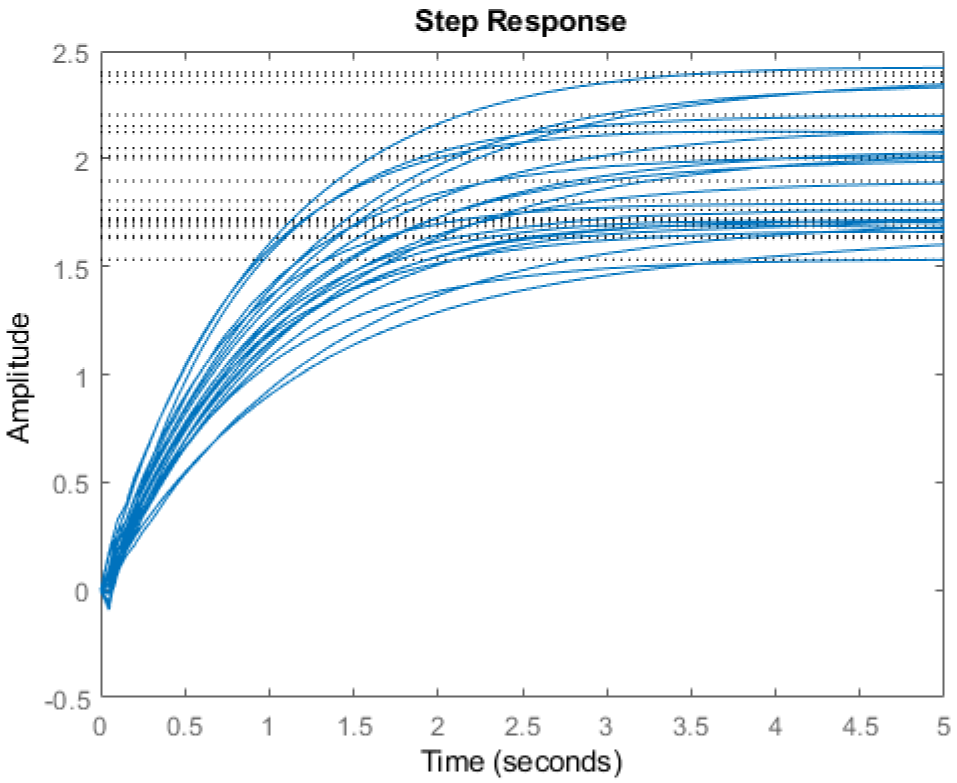
Introduction

The plant model consists of a first-order system with uncertain gain and time constant in series with a mildly underdamped resonance and significant unmodeled dynamics. The uncertain variables are specified using `ureal` and `ultidyn` and the uncertain plant model `P` is constructed as a product of simple transfer functions:

```
gamma = ureal('gamma',2,'Perc',30);  % uncertain gain
tau = ureal('tau',1,'Perc',30);      % uncertain time-constant
wn = 50; xi = 0.25;
P = tf(gamma,[tau 1]) * tf(wn^2,[1 2*xi*wn wn^2]);
% Add unmodeled dynamics
delta = ultidyn('delta',[1 1],'SampleStateDim',5,'Bound',1);
W = makeweight(0.1,20,10);
P = P * (1+W*delta);
```

A collection of step responses for randomly sampled uncertainty values illustrates the plant variability.

```
step(P,5)
```



Covering the Uncertain Model

The uncertain plant model P contains 3 uncertain elements. For feedback design purposes, it is often desirable to simplify the uncertainty model while approximately retaining its overall variability. This is one use of the command ucover. This command takes an array of LTI responses Pa and a nominal response Pn and models the difference Pa-Pn as multiplicative uncertainty in the system dynamics (ultidyn).

To use ucover, first map the uncertain model P into a family of LTI responses using usample. This command samples the uncertain elements in an uncertain system. It returns an array of LTI models where each model representing one possible behavior of the uncertain system. In this example, generate 60 sample values of P:

```
rng('default');     % the random number generator is seeded for repeatability
Parray = usample(P,60);
```

Next, use ucover to cover all behaviors in Parray by a simple uncertain model of the form

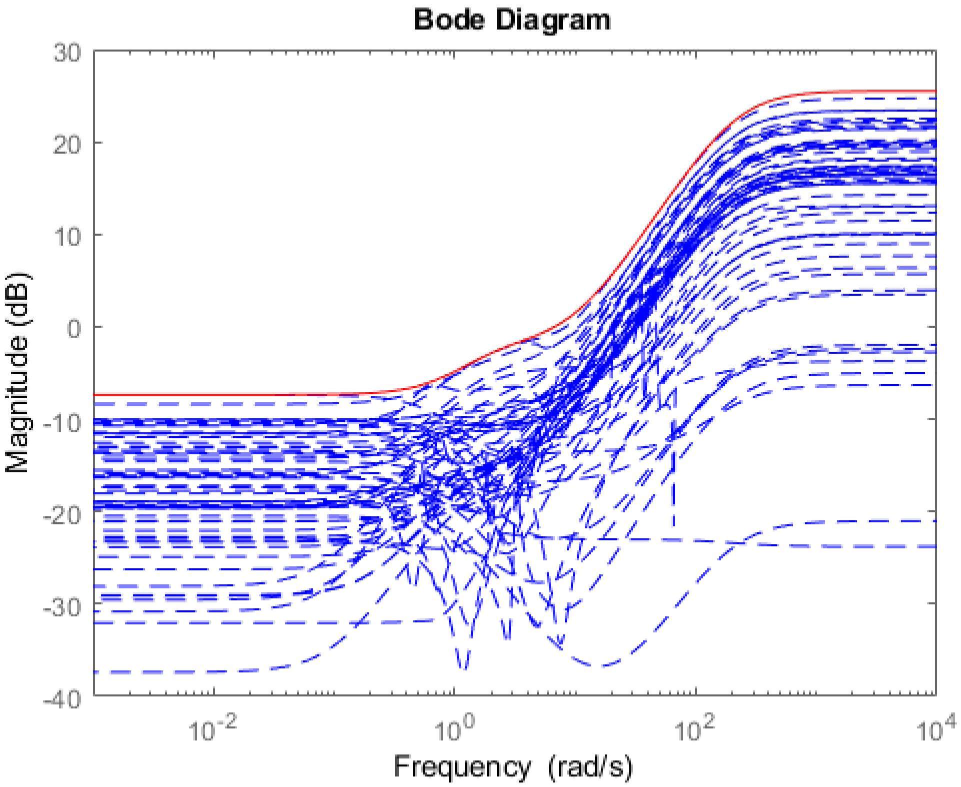
```
Usys = Pn * (1 + Wt * Delta)
```

where all the uncertainty is concentrated in the "unmodeled dynamics" component Delta (a ultidyn object). Choose the nominal value of P as center Pn of the cover, and use a 2nd-order shaping filter Wt to capture how the relative gap between Parray and Pn varies with frequency.

```
Pn = P.NominalValue;
orderWt = 2;
Parrayg = frd(Parray,logspace(-3,3,60));
[Usys,Info] = ucover(Parrayg,Pn,orderWt,'in');
```

Verify that the filter magnitude (in red) "covers" the relative variations of the plant frequency response (in blue).

```
Wt = Info.W1;
bodemag((Pn-Parray)/Pn,'b--',Wt,'r')
```



Creating the Open-Loop Design Model

To design a robust controller for the uncertain plant `P`, choose a target closed-loop bandwidth `desBW` and perform a sensitivity-minimization design using the simplified uncertainty model `Usys`. The control structure is shown in Figure 1. The main signals are the disturbance `d`, the measurement noise `n`, the control signal `u`, and the plant output `y`. The filters `Wperf` and `Wnoise` reflect the frequency content of the disturbance and noise signals, or equivalently, the frequency bands where good disturbance and noise rejection properties are needed.

Our goal is to keep `y` close to zero by rejecting the disturbance `d` and minimizing the impact of the measurement noise `n`. Equivalently, we want to design a controller that keeps the gain from `[d;n]` to `y` "small." Note that

```
y  = Wperf * 1/(1+PC) * d + Wnoise * PC/(1+PC) * n
```

so the transfer function of interest consists of performance- and noise-weighted versions of the sensitivity function 1/(1+PC) and complementary sensitivity function PC/(1+PC).

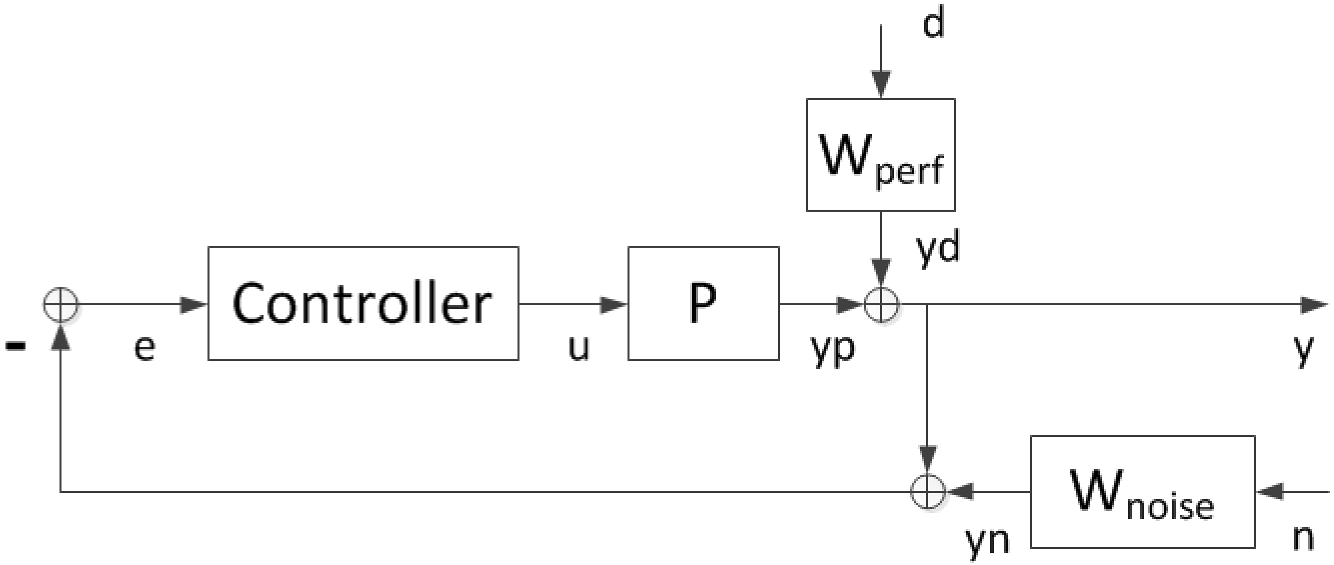


Figure 1: Control Structure.

Choose the performance weighting function `Wperf` as a first-order low-pass filter with magnitude greater than 1 at frequencies below the desired closed-loop bandwidth:

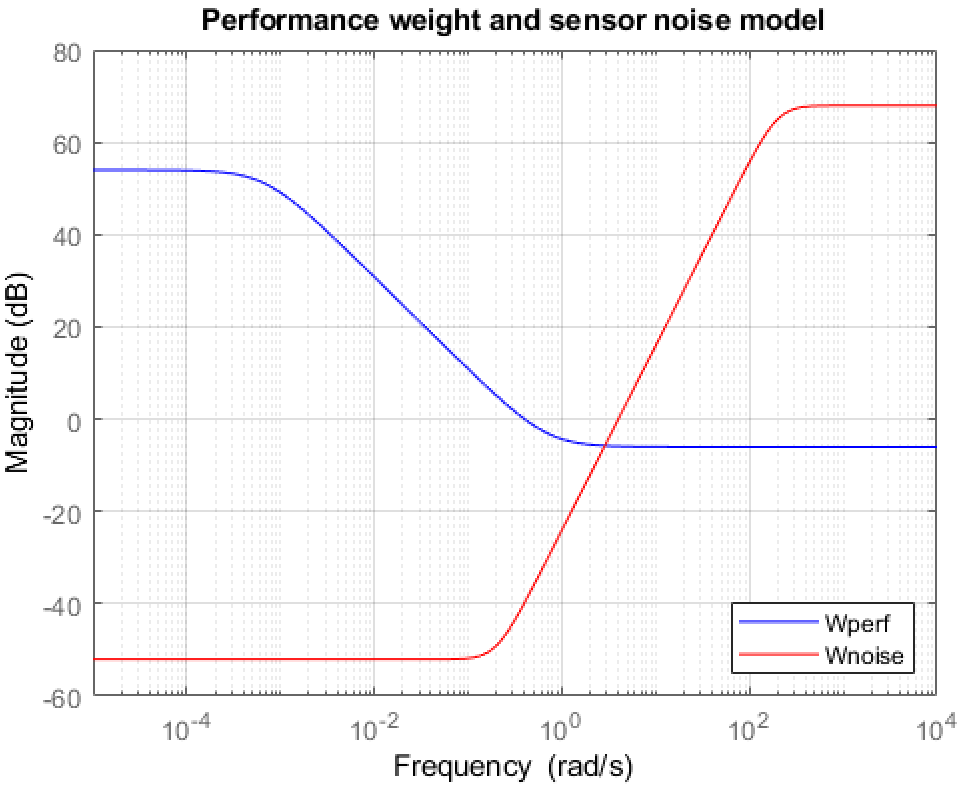
```
desBW = 0.4;
Wperf = makeweight(500,desBW,0.5);
```

To limit the controller bandwidth and induce roll off beyond the desired bandwidth, use a sensor noise model `Wnoise` with magnitude greater than 1 at frequencies greater than `10*desBW`:

```
Wnoise = 0.0025 * tf([25 7 1],[2.5e-5 .007 1]);
```

Plot the magnitude profiles of `Wperf` and `Wnoise`:

```
bodemag(Wperf,'b',Wnoise,'r'), grid
title('Performance weight and sensor noise model')
legend('Wperf','Wnoise','Location','SouthEast')
```



Next build the open-loop interconnection of Figure 1:

```
Usys.InputName = 'u'; Usys.OutputName = 'yp';
Wperf.InputName = 'd'; Wperf.OutputName = 'yd';
Wnoise.InputName = 'n'; Wnoise.OutputName = 'yn';

sumy = sumblk('y = yp + yd');
sume = sumblk('e = -y - yn');

M = connect(Usys,Wperf,Wnoise,sumy,sume,{'d','n','u'},{'y','e'});
```

First Design: Low Bandwidth Requirement

The controller design is carried out with the automated robust design command `musyn`. The uncertain open-loop model is given by `M`.

```
[ny,nu] = size(Usys);
[K1,muBound] = musyn(M,ny,nu);
```

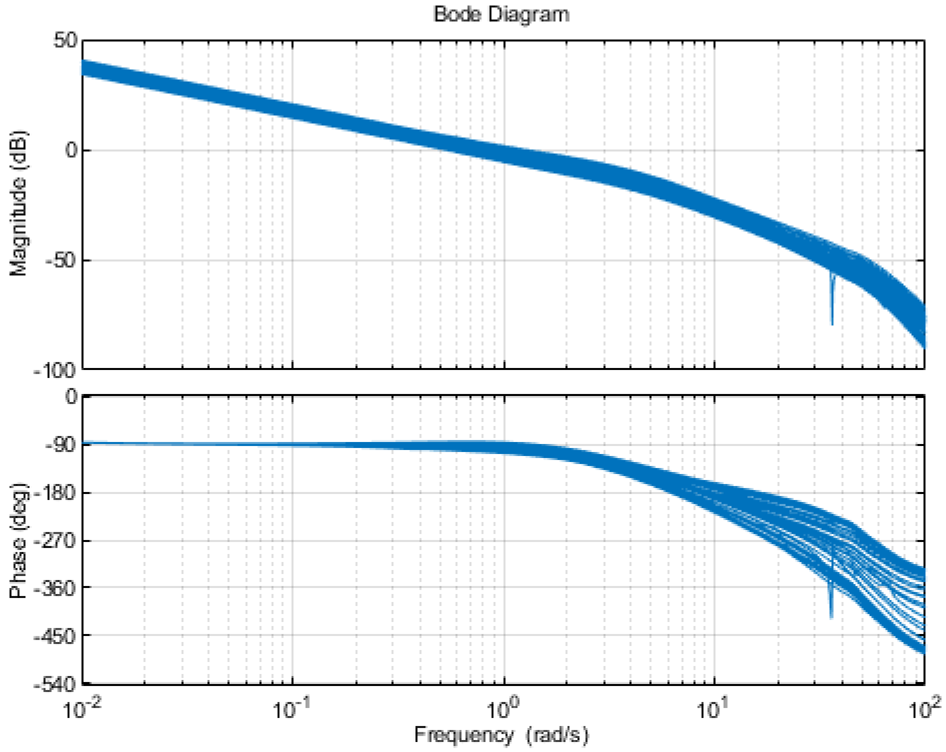
```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step       Peak MU       D Fit             D
    1           223.6        100.4         101.4             2
    2           20.15        1.759         1.774            10
```

```
     3          0.9756       0.9678       0.9783            8
     4          0.9284       0.9284       0.9321            6
     5          0.9117       0.9118       0.9201           10
     6          0.9015       0.9015       0.9038           10
     7          0.8997       0.8997       0.9001           10
     8          0.8967       0.8967       0.8992            8

Best achieved robust performance: 0.897
```

The robust performance muBound is a positive scalar. If it is near 1, then the design is successful and the desired and effective closed-loop bandwidths match closely. As a rule of thumb, if muBound is less than 0.85, then the achievable performance can be improved. When muBound is greater than 1.2, then the desired closed-loop bandwidth is not achievable for the given amount of plant uncertainty.

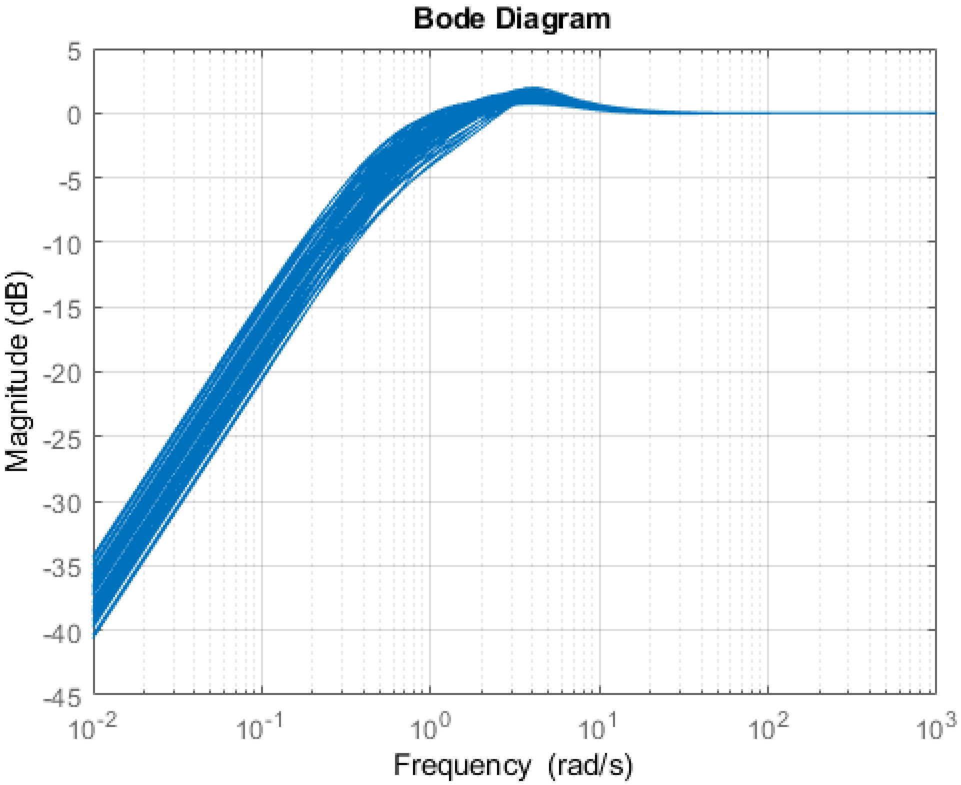
Since, here, muBound is approximately 0.9, the objectives are met, but could ultimately be improved upon. For validation purposes, create Bode plots of the open-loop response for different values of the uncertainty and note the typical zero-dB crossover frequency and phase margin:

```
opt = bodeoptions;
opt.PhaseMatching = 'on';
opt.Grid = 'on';

bodeplot(Parray*K1,{1e-2,1e2},opt);
```



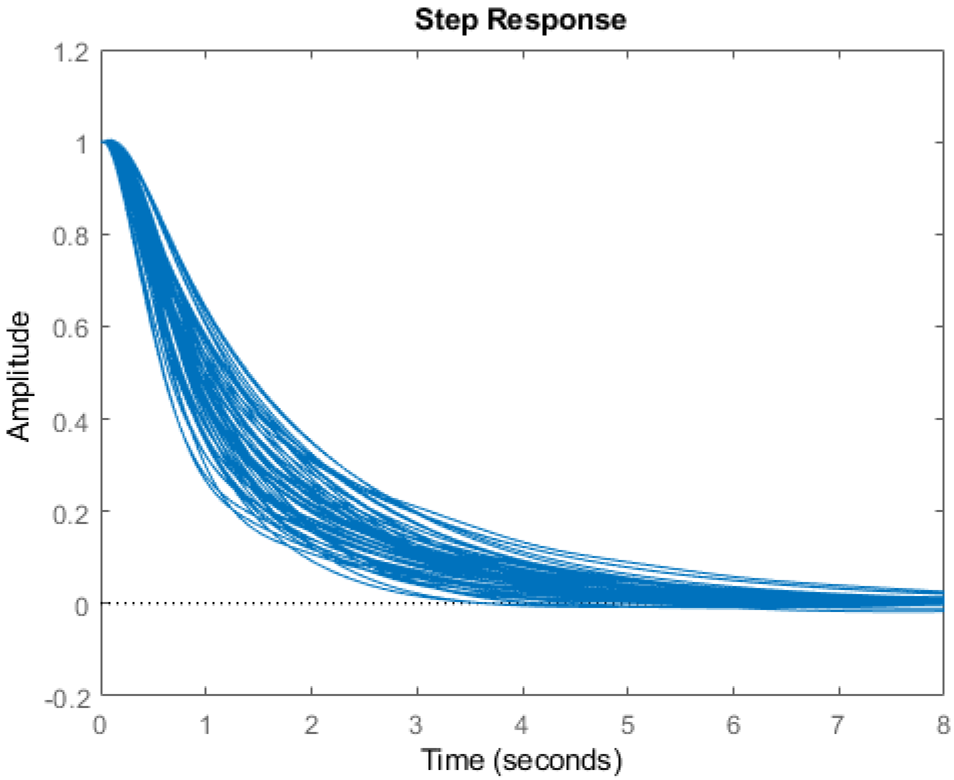
Randomized closed-loop Bode plots confirm a closed-loop disturbance rejection bandwidth of approximately 0.4 rad/s.

```
S1 = feedback(1,Parray*K1);  % sensitivity to output disturbance
bodemag(S1,{1e-2,1e3}), grid
```



Finally, compute and plot the closed-loop responses to a step disturbance at the plant output. These are consistent with the desired closed-loop bandwidth of 0.4, with settling times approximately 7 seconds.

```
step(S1,8);
```



In this naive design strategy, we have correlated the noise bandwidth with the desired closed-loop bandwidth. This simply helps limit the controller bandwidth. A fair perspective is that this approach focuses on output disturbance attenuation in the face of plant model uncertainty. Sensor noise is not truly addressed. Problems with considerable amounts of sensor noise would be dealt with in a different manner.

Second Design: Higher Bandwidth Requirement

Let's redo the design for a higher target bandwidth and adjusting the noise bandwidth as well.

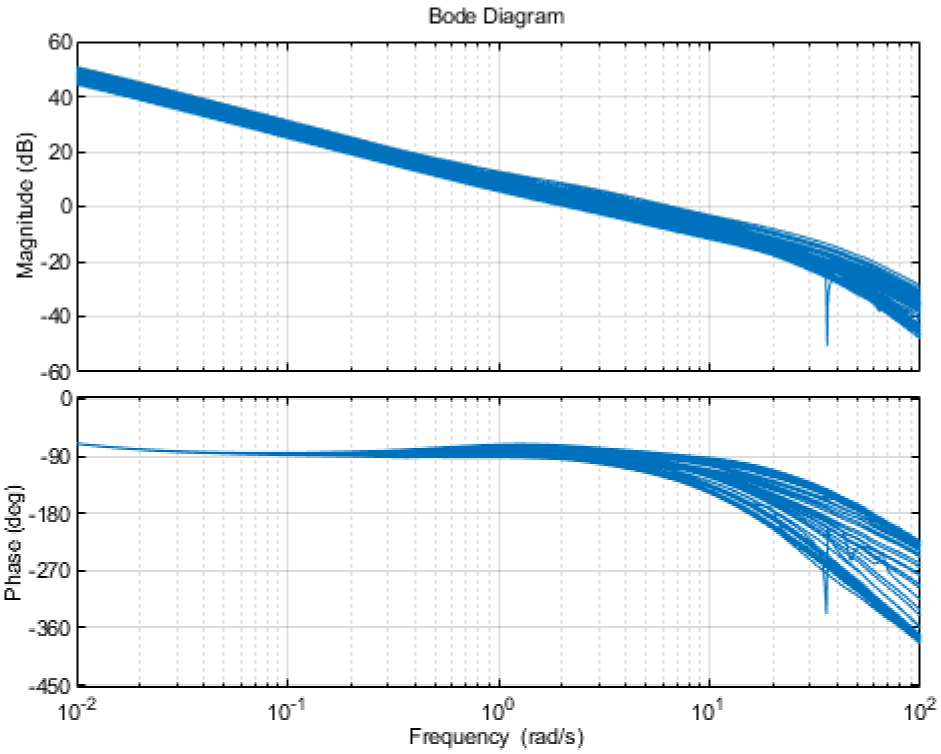
```
desBW = 2;
Wperf = makeweight(500,desBW,0.5);
Wperf.InputName = 'd'; Wperf.OutputName = 'yd';
Wnoise = 0.0025 * tf([1 1.4 1],[1e-6 0.0014 1]);
Wnoise.InputName = 'n'; Wnoise.OutputName = 'yn';

M = connect(Usys,Wperf,Wnoise,sumy,sume,{'d','n','u'},{'y','e'});
[K2,muBound2] = musyn(M,ny,nu);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter             K Step       Peak MU       D Fit             D
    1               223.6        100.5        101.4             2
```

```
      2            20.17        2.116        2.135            10
      3            1.254        1.249        1.261            10
      4            1.178        1.178        1.192            10
      5            1.154        1.154        1.159            10
      6            1.141        1.141        1.144            10
      7            1.132        1.132        1.135            10
      8            1.124        1.124        1.125            10
      9            1.121        1.121        1.125             8
     10            1.114        1.114        1.124            10

Best achieved robust performance: 1.11
```

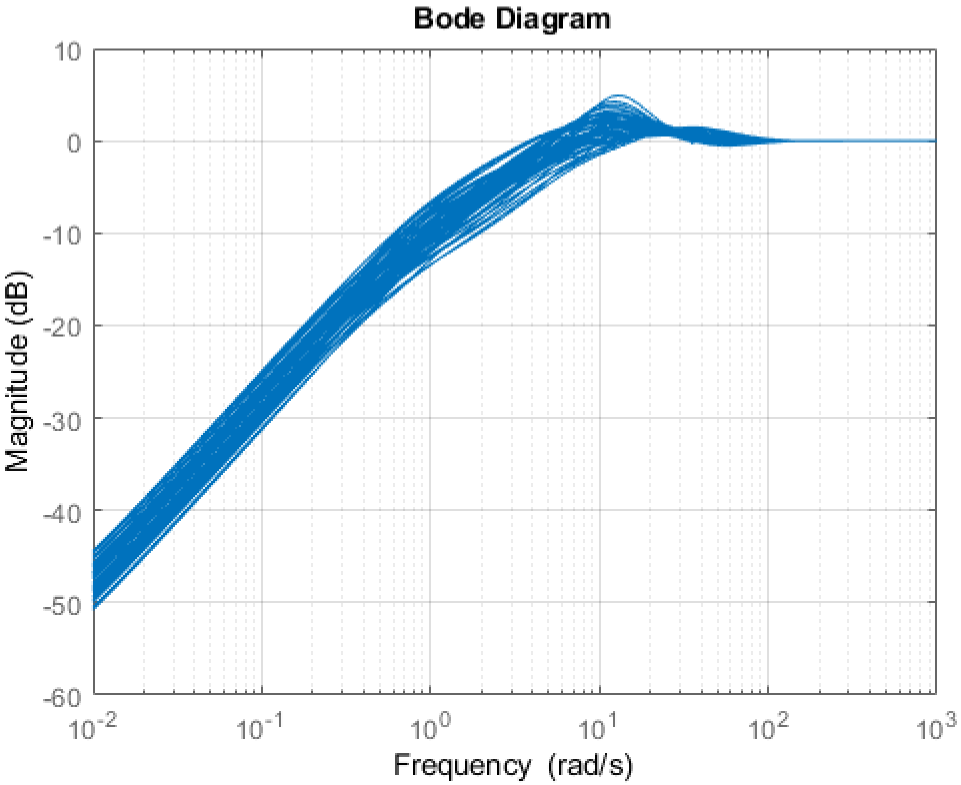
With a robust performance of about 1.1, this design achieves a good tradeoff between performance goals and plant uncertainty. Open-loop Bode plots confirm a fairly robust design with decent phase margins, but not as good as the lower bandwidth design.

```
bodeplot(Parray*K2,{1e-2,1e2},opt)
```



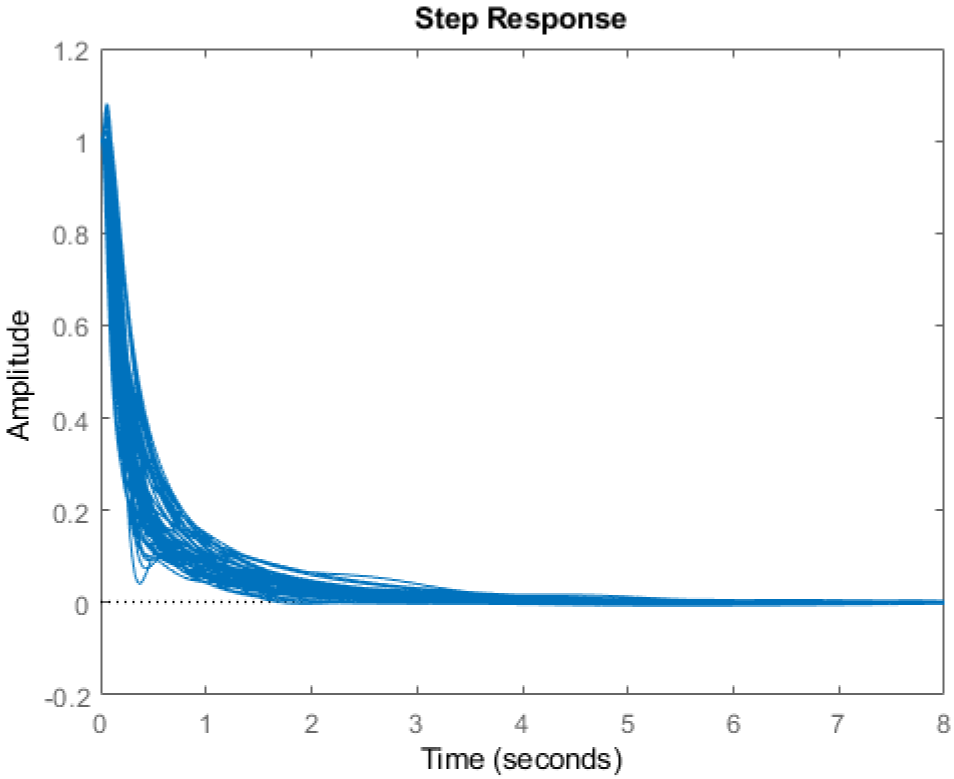
Randomized closed-loop Bode plots confirm a closed-loop bandwidth of approximately 2 rad/s. The frequency response has a bit more peaking than was seen in the lower bandwidth design, due to the increased uncertainty in the model at this frequency. Since the Robust Performance mu-value was 1.1, we expected some degradation in the robustness of the performance objectives over the lower bandwidth design.

```
S2 = feedback(1,Parray*K2);
bodemag(S2,{1e-2,1e3}), grid
```



Closed-loop step disturbance responses further illustrate the higher bandwidth response, with reasonable robustness across the plant model variability.

```
step(S2,8);
```



Third Design: Very Aggressive Bandwidth Requirement

Redo the design once more with an extremely optimistic closed-loop bandwidth goal of 15 rad/s.

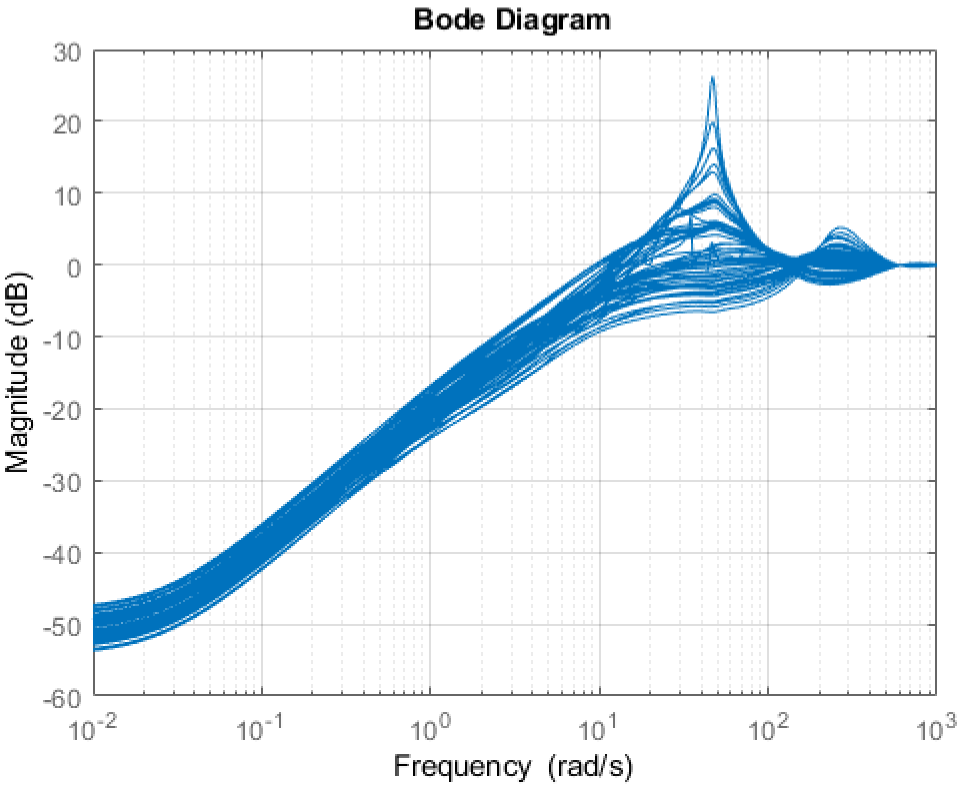
```
desBW = 15;
Wperf = makeweight(500,desBW,0.5);
Wperf.InputName = 'd'; Wperf.OutputName = 'yd';
Wnoise = 0.0025 * tf([0.018 0.19 1],[0.018e-6 0.19e-3 1]);
Wnoise.InputName = 'n'; Wnoise.OutputName = 'yn';

M = connect(Usys,Wperf,Wnoise,sumy,sume,{'d','n','u'},{'y','e'});
[K3,muBound3] = musyn(M,ny,nu);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter         K Step       Peak MU       D Fit             D
    1           223.6         100.9       101.8             2
    2           20.26         3.618       3.649             8
    3           2.189         2.189       2.215            10
    4           1.997         1.997       2.014            10
    5           1.919         1.919       1.933            10
    6           1.873         1.873       1.932             6
    7            1.85          1.85       1.872            10
    8           1.826         1.826       1.845             8
```

```
     9          1.818        1.818        1.83            8
    10          1.811        1.811       1.818            8

Best achieved robust performance: 1.81
```

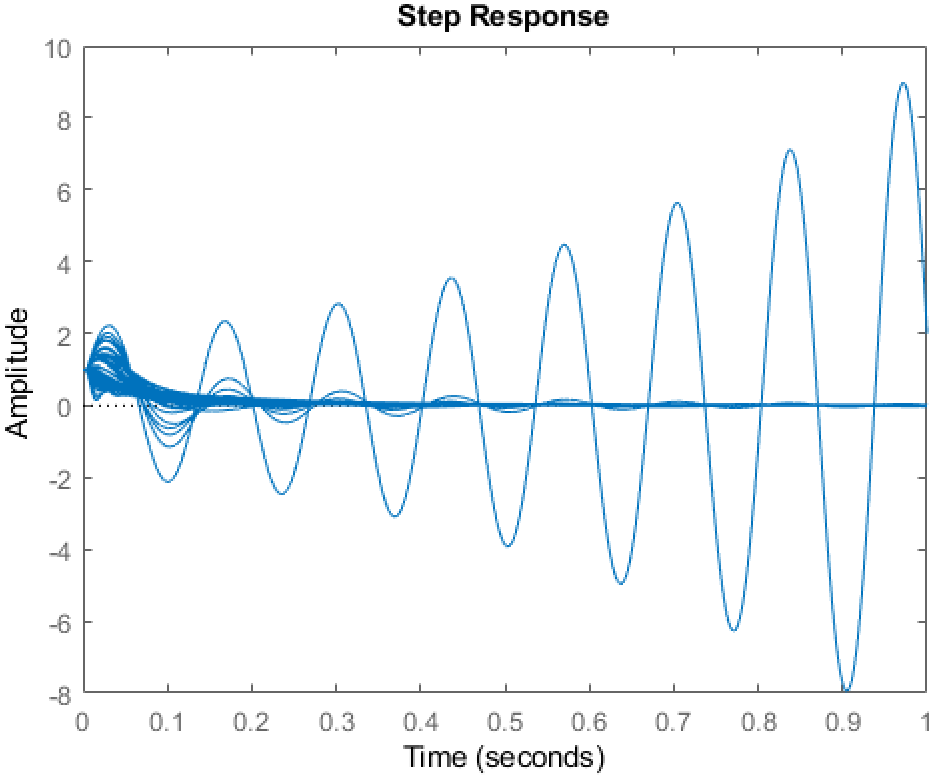
Since the robust performance is greater than 1.8, the closed-loop performance goals are not achieved under plant uncertainties. The frequency responses of the closed-loop system have higher peaks indicating the poor performance of the designed controller.

```
S3 = feedback(1,Parray*K3);
bodemag(S3,{1e-2,1e3}), grid
```



Similarly, step responses under uncertainties illustrate the poor closed-loop performance.

```
step(S3,1);
```



Robust Stability Calculations

The Bode and Step response plots shown above are generated from samples of the uncertain plant model P. We can use the uncertain model directly, and assess the robust stability of the three closed-loop systems.

```
ropt = robOptions('Display','on','MussvOptions','sm5');
stabmarg1 = robstab(feedback(P,K1),ropt);
```

```
Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 275% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 276% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 7.69 rad/seconds.
```

```
stabmarg2 = robstab(feedback(P,K2),ropt);
```

```
Computing peak...  Percent completed: 100/100
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 151% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 151% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 17.3 rad/seconds.
```

```
stabmarg3 = robstab(feedback(P,K3),ropt);
```

```
Computing peak...  Percent completed: 100/100
System is not robustly stable for the modeled uncertainty.
 -- It can tolerate up to 83.9% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 84% of the modeled uncertainty.
 -- This perturbation causes an instability at the frequency 79.5 rad/seconds.
```

The robustness analysis reports confirm what we have observed by sampling the closed-loop time and frequency responses. The second design is a good compromise between performance and robustness, and the third design is too aggressive and lacks robustness.

See Also

`musyn` | `robstab` | `ucover`

More About

- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “Robustness and Worst-Case Analysis” on page 2-21
- “Control of a Two-Tank System” on page 3-34

Control of a Two-Tank System

This example shows how to use Robust Control Toolbox™ to design a robust controller (using D-K iteration) and to do robustness analysis on a process control problem. In our example, the plant is a simple two-tank system.

Additional experimental work relating to this system is described by Smith et al. in the following references:

- Smith, R.S., J. Doyle, M. Morari, and A. Skjellum, "A Case Study Using mu: Laboratory Process Control Problem," Proceedings of the 10th IFAC World Congress, vol. 8, pp. 403-415, 1987.
- Smith, R.S, and J. Doyle, "The Two Tank Experiment: A Benchmark Control Problem," in Proceedings American Control Conference, vol. 3, pp. 403-415, 1988.
- Smith, R.S., and J. C. Doyle, "Closed Loop Relay Estimation of Uncertainty Bounds for Robust Control Models," in Proceedings of the 12th IFAC World Congress, vol. 9, pp. 57-60, July 1993.

Plant Description

The plant in our example consists of two water tanks in cascade as shown schematically in Figure 1. The upper tank (tank 1) is fed by hot and cold water via computer-controlled valves. The lower tank (tank 2) is fed by water from an exit at the bottom of tank 1. An overflow maintains a constant level in tank 2. A cold water bias stream also feeds tank 2 and enables the tanks to have different steady-state temperatures.

Our design objective is to control the temperatures of both tanks 1 and 2. The controller has access to the reference commands and the temperature measurements.

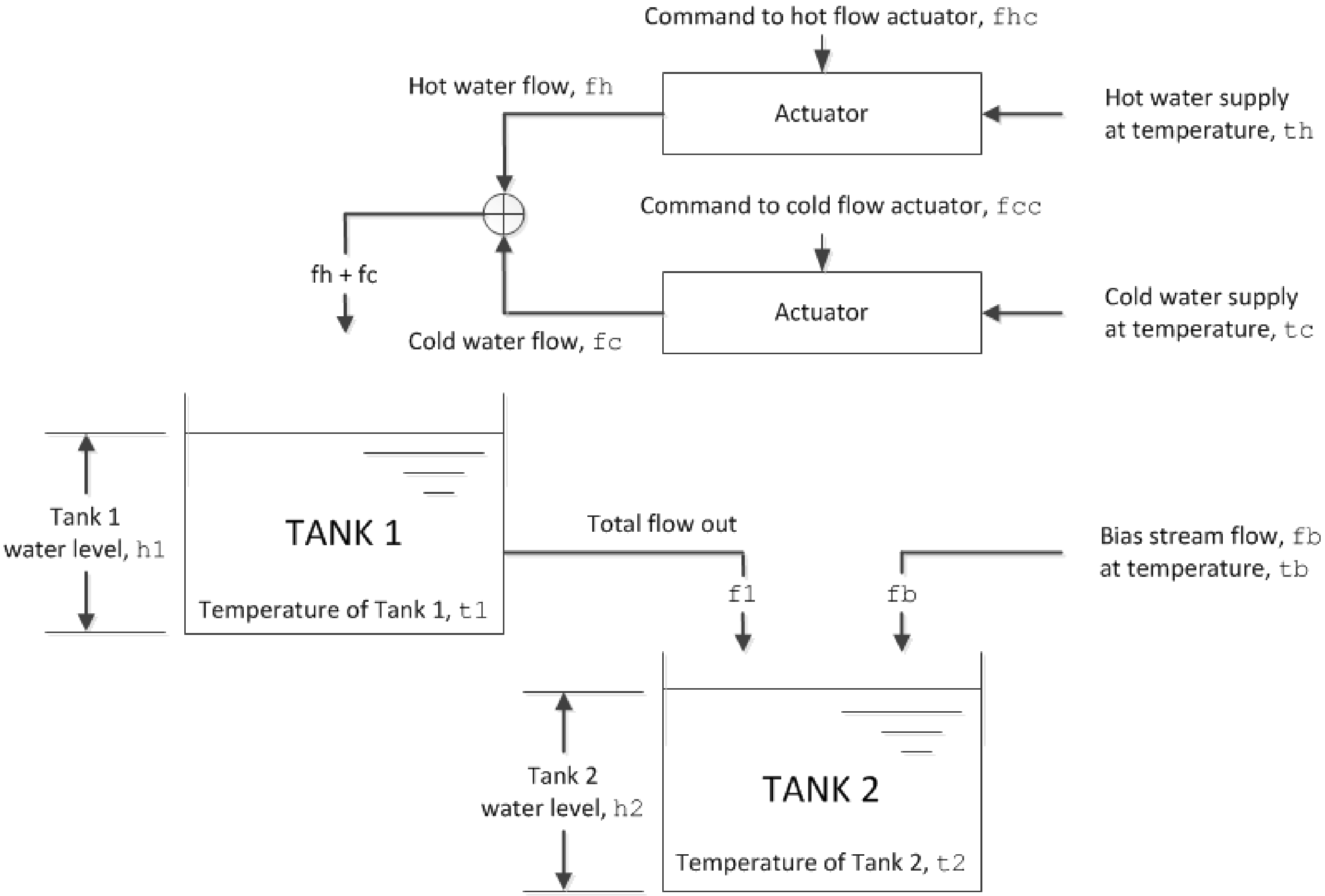

Command to hot flow actuator, fhc
Hot water flow, fh
Actuator
Hot water supply at temperature, th
Command to cold flow actuator, fcc
fh + fc
Actuator
Cold water supply at temperature, tc
Cold water flow, fc
Tank 1 water level, h1
TANK 1
Temperature of Tank 1, t1
Total flow out
f1
fb
Bias stream flow, fb at temperature, tb
Tank 2 water level, h2
TANK 2
Temperature of Tank 2, t2

Figure 1: Schematic diagram of a two-tank system

Tank Variables

Let's give the plant variables the following designations:

- `fhc`: Command to hot flow actuator
- `fh`: Hot water flow into tank 1
- `fcc`: Command to cold flow actuator
- `fc`: Cold water flow into tank 1
- `f1`: Total flow out of tank 1
- `A1`: Cross-sectional area of tank 1
- `h1`: Tank 1 water level
- `t1`: Temperature of tank 1
- `t2`: Temperature of tank 2
- `A2`: Cross-sectional area of tank 2
- `h2`: Tank 2 water level
- `fb`: Flow rate of tank 2 bias stream

- `tb`: Temperature of tank 2 bias stream
- `th`: Hot water supply temperature
- `tc`: Cold water supply temperature

For convenience we define a system of normalized units as follows:

```
  Variable      Unit Name      0 means:            1 means:
  --------      ---------      --------            --------
  temperature   tunit          cold water temp.    hot water temp.
  height        hunit          tank empty          tank full
  flow          funit          zero input flow     max. input flow
```

Using the above units, these are the plant parameters:

```
A1 = 0.0256;    % Area of tank 1 (hunits^2)
A2 = 0.0477;    % Area of tank 2 (hunits^2)
h2 = 0.241;        % Height of tank 2, fixed by overflow (hunits)
fb = 3.28e-5;   % Bias stream flow (hunits^3/sec)
fs = 0.00028;    % Flow scaling (hunits^3/sec/funit)
th = 1.0;        % Hot water supply temp (tunits)
tc = 0.0;        % Cold water supply temp (tunits)
tb = tc;        % Cold bias stream temp (tunits)
alpha = 4876;   % Constant for flow/height relation (hunits/funits)
beta = 0.59;    % Constant for flow/height relation (hunits)
```

The variable fs is a flow-scaling factor that converts the input (0 to 1 funits) to flow in hunits^3/second. The constants alpha and beta describe the flow/height relationship for tank 1:

h1 = alpha*f1-beta.

Nominal Tank Models

We can obtain the nominal tank models by linearizing around the following operating point (all normalized values):

```
h1ss = 0.75;                            % Water level for tank 1
t1ss = 0.75;                            % Temperature of tank 1
f1ss = (h1ss+beta)/alpha;               % Flow tank 1 -> tank 2
fss = [th,tc;1,1]\[t1ss*f1ss;f1ss];
fhss = fss(1);                          % Hot flow
fcss = fss(2);                          % Cold flow
t2ss = (f1ss*t1ss + fb*tb)/(f1ss + fb); % Temperature of tank 2
```

The nominal model for tank 1 has inputs [`fh`; `fc`] and outputs [`h1`; `t1`]:

```
A = [ -1/(A1*alpha),          0;
      (beta*t1ss)/(A1*h1ss),  -(h1ss+beta)/(alpha*A1*h1ss)];

B = fs*[ 1/(A1*alpha),   1/(A1*alpha);
         th/A1,          tc/A1];

C = [ alpha,             0;
      -alpha*t1ss/h1ss,  1/h1ss];

D = zeros(2,2);
tank1nom = ss(A,B,C,D,'InputName',{'fh','fc'},'OutputName',{'h1','t1'});
```

```
clf
step(tank1nom), title('Step responses of Tank 1')
```

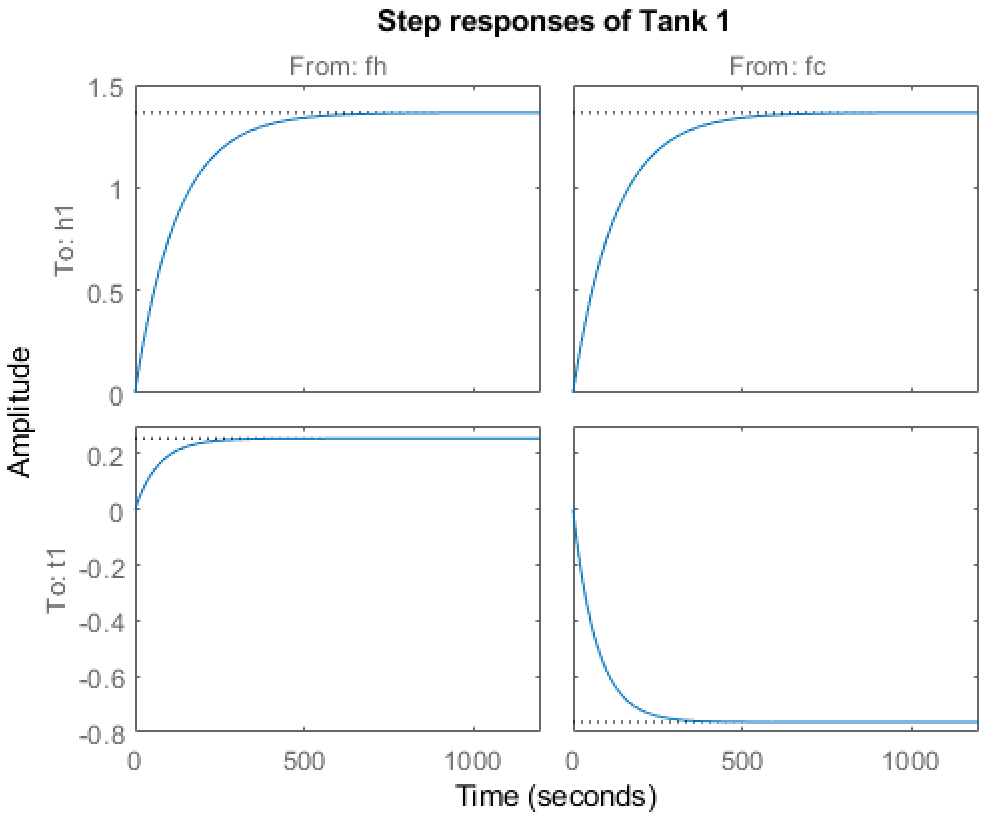


Figure 2: Step responses of Tank 1.

The nominal model for tank 2 has inputs [|h1|;|t1|] and output `t2`:

```
A = -(h1ss + beta + alpha*fb)/(A2*h2*alpha);
B = [ (t2ss+t1ss)/(alpha*A2*h2),  (h1ss + beta)/(alpha*A2*h2) ];
C = 1;
D = zeros(1,2);

tank2nom = ss(A,B,C,D,'InputName',{'h1','t1'},'OutputName','t2');

step(tank2nom), title('Step responses of Tank 2')
```

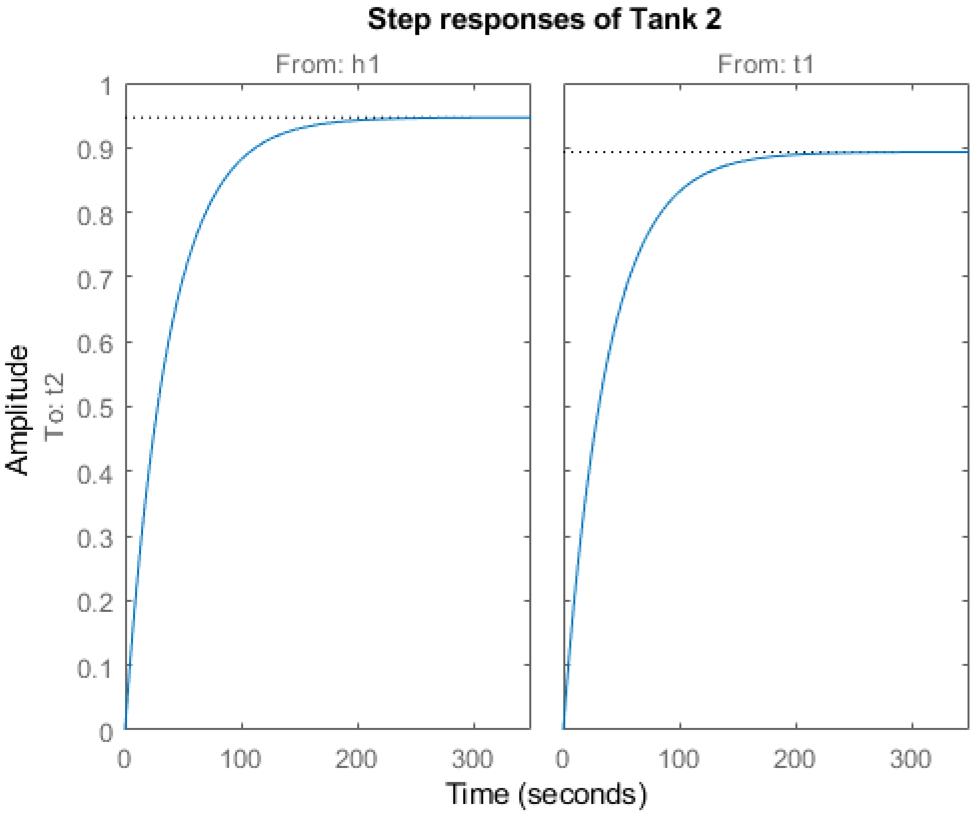


Figure 3: Step responses of Tank 2.

Actuator Models

There are significant dynamics and saturations associated with the actuators, so we'll want to include actuator models. In the frequency range we're using, we can model the actuators as a first order system with rate and magnitude saturations. It is the rate limit, rather than the pole location, that limits the actuator performance for most signals. For a linear model, some of the effects of rate limiting can be included in a perturbation model.

We initially set up the actuator model with one input (the command signal) and two outputs (the actuated signal and its derivative). We'll use the derivative output in limiting the actuation rate when designing the control law.

```
act_BW = 20;        % Actuator bandwidth (rad/sec)
actuator = [ tf(act_BW,[1 act_BW]); tf([act_BW 0],[1 act_BW]) ];
actuator.OutputName = {'Flow','Flow rate'};

bodemag(actuator)
title('Valve actuator dynamics')

hot_act = actuator;
set(hot_act,'InputName','fhc','OutputName',{'fh','fh_rate'});
cold_act =actuator;
set(cold_act,'InputName','fcc','OutputName',{'fc','fc_rate'});
```

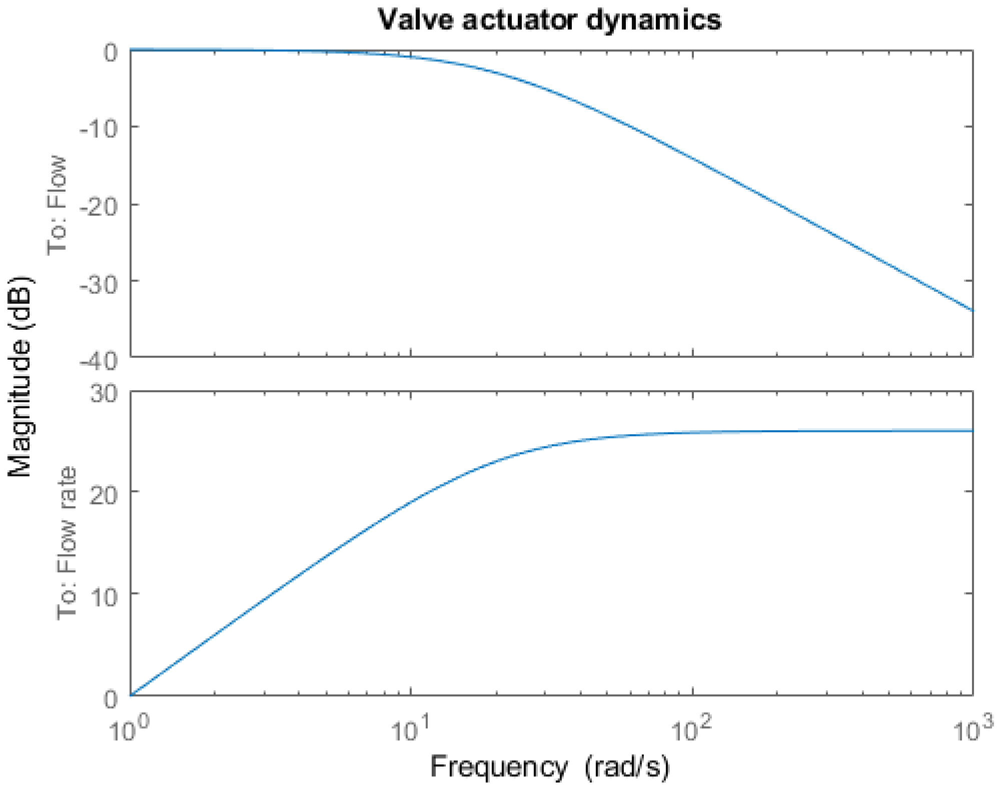


Figure 4: Valve actuator dynamics.

Anti-Aliasing Filters

All measured signals are filtered with fourth-order Butterworth filters, each with a cutoff frequency of 2.25 Hz.

```
fbw = 2.25;        % Anti-aliasing filter cut-off (Hz)
filter = mkfilter(fbw,4,'Butterw');
h1F = filter;
t1F = filter;
t2F = filter;
```

Uncertainty on Model Dynamics

Open-loop experiments reveal some variability in the system responses and suggest that the linear models are good at low frequency. If we fail to take this information into account during the design, our controller might perform poorly on the real system. For this reason, we will build an uncertainty model that matches our estimate of uncertainty in the physical system as closely as possible. Because the amount of model uncertainty or variability typically depends on frequency, our uncertainty model involves frequency-dependent weighting functions to normalize modeling errors across frequency.

For example, open-loop experiments indicate a significant amount of dynamic uncertainty in the t1 response. This is due primarily to mixing and heat loss. We can model it as a multiplicative (relative) model error Delta2 at the t1 output. Similarly, we can add multiplicative model errors Delta1 and Delta3 to the h1 and t2 outputs as shown in Figure 5.

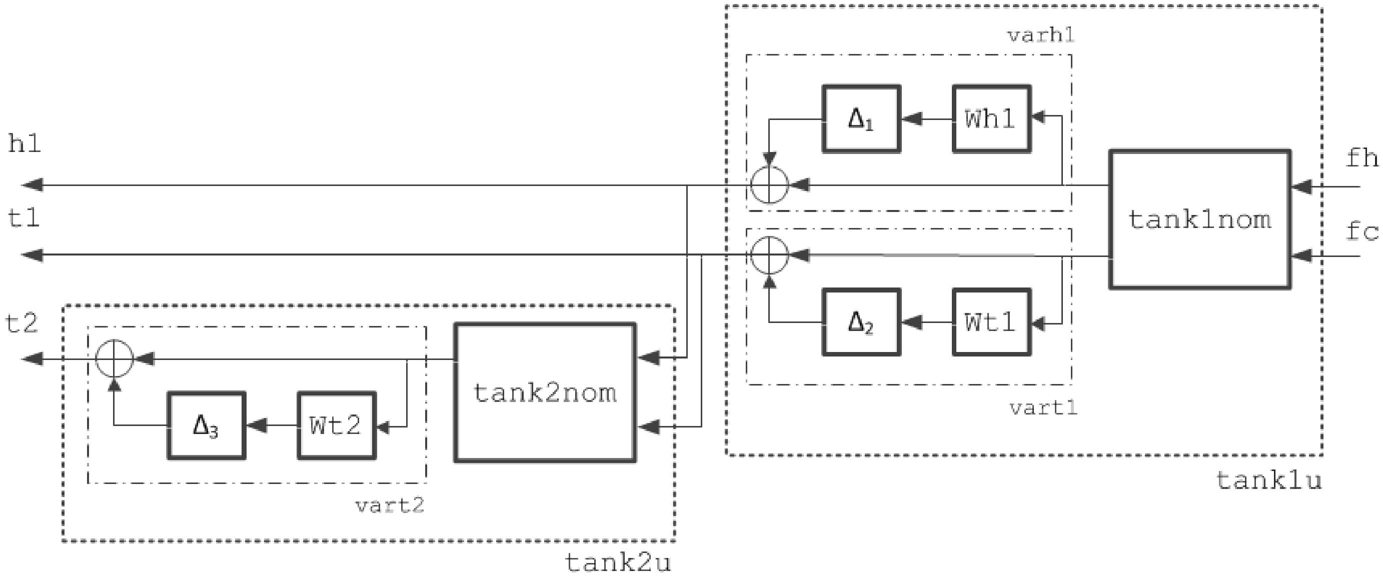


Figure 5: Schematic representation of a perturbed, linear two-tank system.

To complete the uncertainty model, we quantify how big the modeling errors are as a function of frequency. While it's difficult to determine precisely the amount of uncertainty in a system, we can look for rough bounds based on the frequency ranges where the linear model is accurate or poor, as in these cases:

- The nominal model for `h1` is very accurate up to at least 0.3 Hz.
- Limit-cycle experiments in the `t1` loop suggest that uncertainty should dominate above 0.02 Hz.
- There are about 180 degrees of additional phase lag in the `t1` model at about 0.02 Hz. There is also a significant gain loss at this frequency. These effects result from the unmodeled mixing dynamics.
- Limit cycle experiments in the `t2` loop suggest that uncertainty should dominate above 0.03 Hz.

This data suggests the following choices for the frequency-dependent modeling error bounds.

```
Wh1 = 0.01+tf([0.5,0],[0.25,1]);
Wt1 = 0.1+tf([20*h1ss,0],[0.2,1]);
Wt2 = 0.1+tf([100,0],[1,21]);

clf
bodemag(Wh1,Wt1,Wt2), title('Relative bounds on modeling errors')
legend('h1 dynamics','t1 dynamics','t2 dynamics','Location','NorthWest')
```

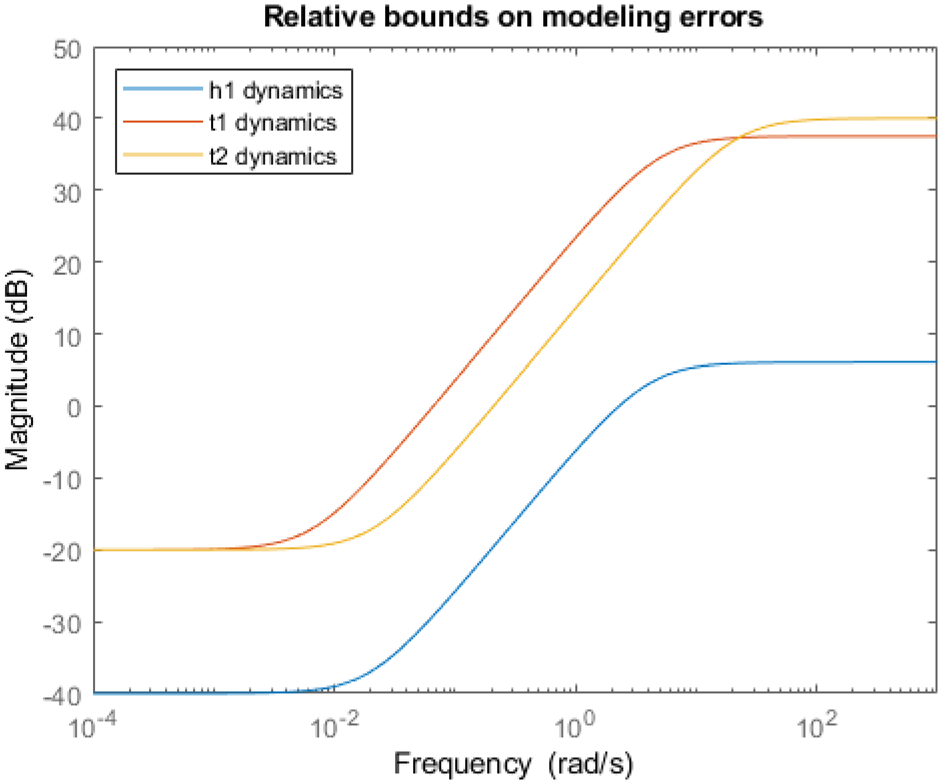


Figure 6: Relative bounds on modeling errors.

Now, we're ready to build uncertain tank models that capture the modeling errors discussed above.

```
% Normalized error dynamics
delta1 = ultidyn('delta1',[1 1]);
delta2 = ultidyn('delta2',[1 1]);
delta3 = ultidyn('delta3',[1 1]);

% Frequency-dependent variability in h1, t1, t2 dynamics
varh1 = 1+delta1*Wh1;
vart1 = 1+delta2*Wt1;
vart2 = 1+delta3*Wt2;

% Add variability to nominal models
tank1u = append(varh1,vart1)*tank1nom;
tank2u = vart2*tank2nom;

tank1and2u = [0 1; tank2u]*tank1u;
```

Next, we randomly sample the uncertainty to see how the modeling errors might affect the tank responses

```
step(tank1u,1000), title('Variability in responses due to modeling errors (Tank 1)')
```

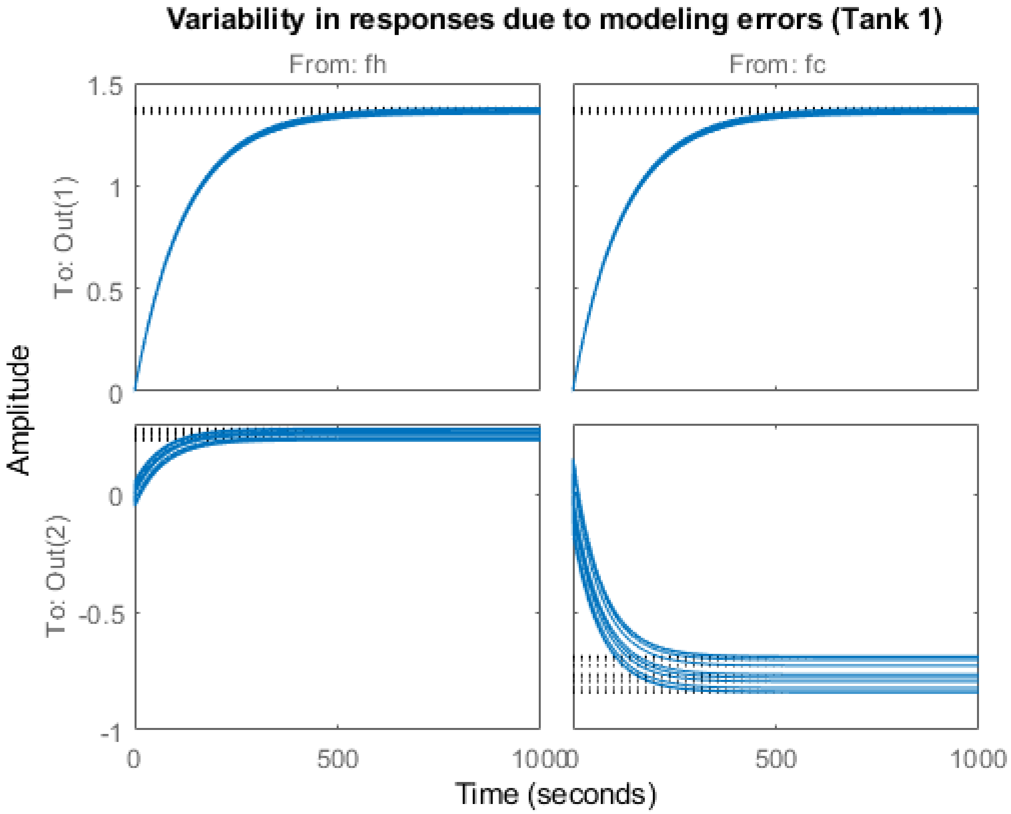


Figure 7: Variability in responses due to modeling errors (Tank 1).

Setting up a Controller Design

Now let's look at the control design problem. We're interested in tracking setpoint commands for `t1` and `t2`. To take advantage of H-infinity design algorithms, we must formulate the design as a closed-loop gain minimization problem. To do so, we select weighting functions that capture the disturbance characteristics and performance requirements to help normalize the corresponding frequency-dependent gain constraints.

Here is a suitable weighted open-loop transfer function for the two-tank problem:

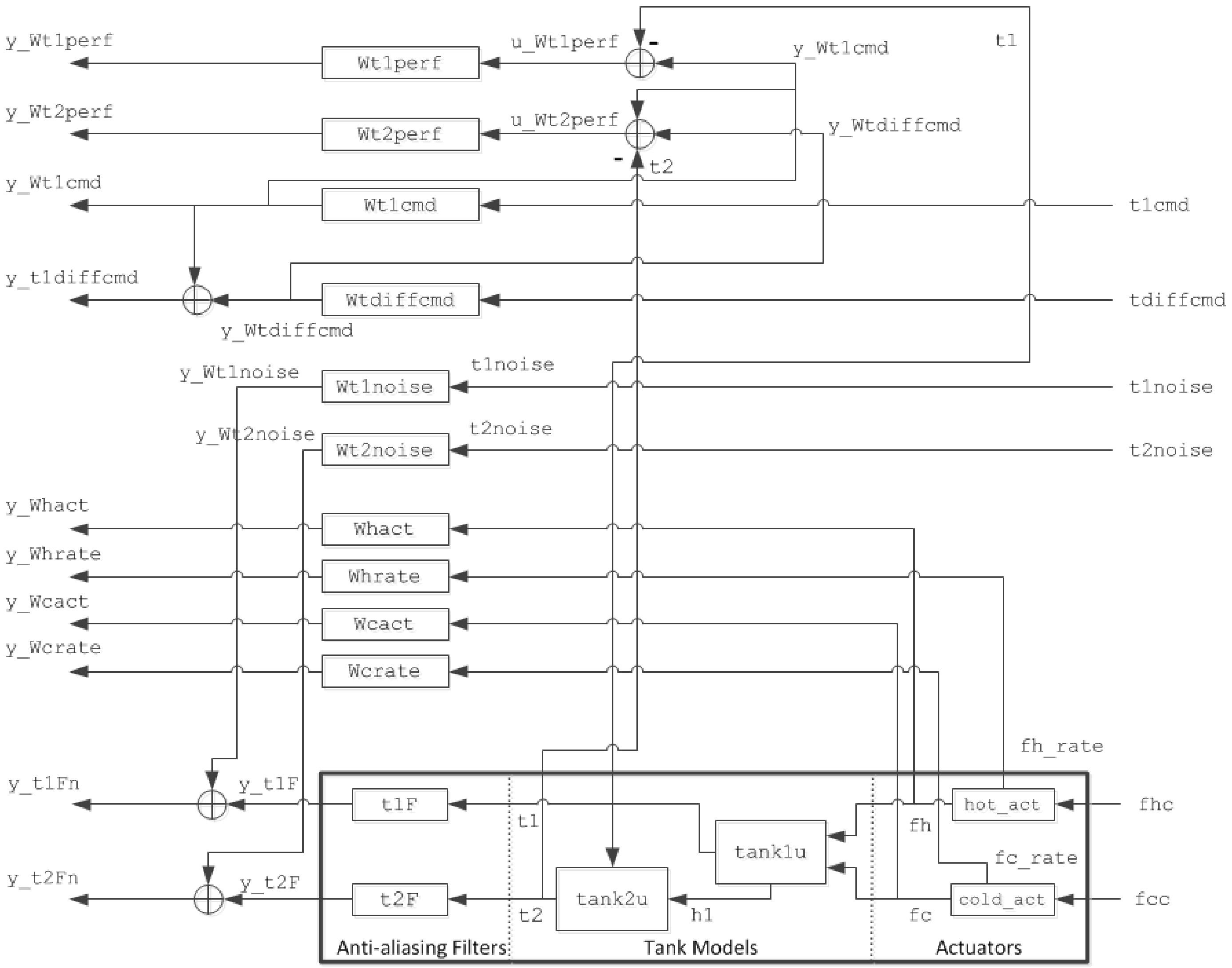


Figure 8: Control design interconnection for two-tank system.

Next, we select weights for the sensor noises, setpoint commands, tracking errors, and hot/cold actuators.

The sensor dynamics are insignificant relative to the dynamics of the rest of the system. This is not true of the sensor noise. Potential sources of noise include electronic noise in thermocouple compensators, amplifiers, and filters, radiated noise from the stirrers, and poor grounding. We use smoothed FFT analysis to estimate the noise level, which suggests the following weights:

```
Wh1noise = zpk(0.01);  % h1 noise weight
Wt1noise = zpk(0.03);  % t1 noise weight
Wt2noise = zpk(0.03);  % t2 noise weight
```

The error weights penalize setpoint tracking errors on `t1` and `t2`. We'll pick first-order low-pass filters for these weights. We use a higher weight (better tracking) for `t1` because physical considerations lead us to believe that `t1` is easier to control than `t2`.

```
Wt1perf = tf(100,[400,1]);    % t1 tracking error weight
Wt2perf = tf(50,[800,1]);    % t2 tracking error weight

clf
bodemag(Wt1perf,Wt2perf)
title('Frequency-dependent penalty on setpoint tracking errors')
legend('t1','t2')
```

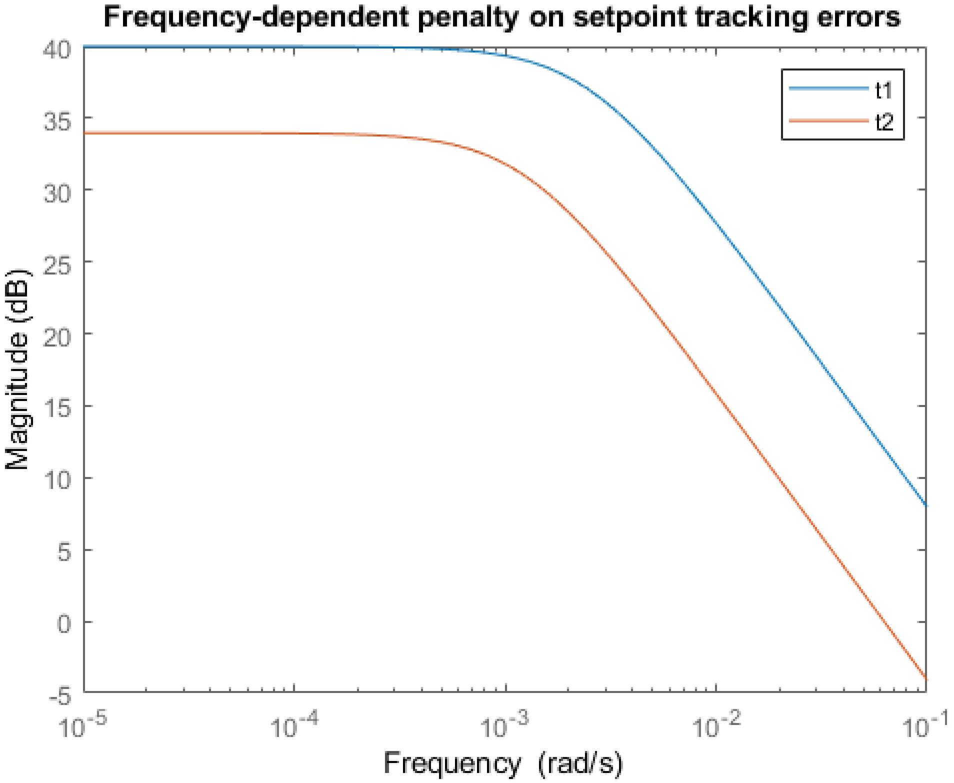


Figure 9: Frequency-dependent penalty on setpoint tracking errors.

The reference (setpoint) weights reflect the frequency contents of such commands. Because the majority of the water flowing into tank 2 comes from tank 1, changes in `t2` are dominated by changes in `t1`. Also `t2` is normally commanded to a value close to `t1`. So it makes more sense to use setpoint weighting expressed in terms of `t1` and `t2-t1`:

```
  t1cmd = Wt1cmd * w1

  t2cmd = Wt1cmd * w1 + Wtdiffcmd * w2
```

where w1, w2 are white noise inputs. Adequate weight choices are:

```
Wt1cmd = zpk(0.1);                % t1 input command weight
Wtdiffcmd = zpk(0.01);            % t2 - t1  input command weight
```

Finally, we would like to penalize both the amplitude and the rate of the actuator. We do this by weighting `fhc` (and `fcc`) with a function that rolls up at high frequencies. Alternatively, we can create an actuator model with `fh` and d|fh|/dt as outputs, and weight each output separately with constant weights. This approach has the advantage of reducing the number of states in the weighted open-loop model.

```
Whact =  zpk(0.01);  % Hot actuator penalty
Wcact =  zpk(0.01);  % Cold actuator penalty

Whrate = zpk(50);    % Hot actuator rate penalty
Wcrate = zpk(50);    % Cold actuator rate penalty
```

Building a Weighted Open-Loop Model

Now that we have modeled all plant components and selected our design weights, we'll use the `connect` function to build an uncertain model of the weighted open-loop model shown in Figure 8.

```
inputs = {'t1cmd', 'tdiffcmd', 't1noise', 't2noise', 'fhc', 'fcc'};
outputs = {'y_Wt1perf', 'y_Wt2perf', 'y_Whact', 'y_Wcact', ...
             'y_Whrate', 'y_Wcrate', 'y_Wt1cmd', 'y_t1diffcmd', ...
                                           'y_t1Fn', 'y_t2Fn'};

hot_act.InputName = 'fhc'; hot_act.OutputName = {'fh' 'fh_rate'};
cold_act.InputName = 'fcc'; cold_act.OutputName = {'fc' 'fc_rate'};

tank1and2u.InputName = {'fh','fc'};
tank1and2u.OutputName = {'t1','t2'};

t1F.InputName = 't1'; t1F.OutputName = 'y_t1F';
t2F.InputName = 't2'; t2F.OutputName = 'y_t2F';

Wt1cmd.InputName = 't1cmd'; Wt1cmd.OutputName = 'y_Wt1cmd';
Wtdiffcmd.InputName = 'tdiffcmd'; Wtdiffcmd.OutputName = 'y_Wtdiffcmd';

Whact.InputName = 'fh'; Whact.OutputName = 'y_Whact';
Wcact.InputName = 'fc'; Wcact.OutputName = 'y_Wcact';

Whrate.InputName = 'fh_rate'; Whrate.OutputName = 'y_Whrate';
Wcrate.InputName = 'fc_rate'; Wcrate.OutputName = 'y_Wcrate';

Wt1perf.InputName = 'u_Wt1perf'; Wt1perf.OutputName = 'y_Wt1perf';
Wt2perf.InputName = 'u_Wt2perf'; Wt2perf.OutputName = 'y_Wt2perf';

Wt1noise.InputName = 't1noise'; Wt1noise.OutputName = 'y_Wt1noise';
Wt2noise.InputName = 't2noise'; Wt2noise.OutputName = 'y_Wt2noise';

sum1 = sumblk('y_t1diffcmd = y_Wt1cmd + y_Wtdiffcmd');
sum2 = sumblk('y_t1Fn = y_t1F + y_Wt1noise');
sum3 = sumblk('y_t2Fn = y_t2F + y_Wt2noise');
sum4 = sumblk('u_Wt1perf = y_Wt1cmd - t1');
sum5 = sumblk('u_Wt2perf = y_Wtdiffcmd + y_Wt1cmd - t2');

% This produces the uncertain state-space model
P = connect(tank1and2u,hot_act,cold_act,t1F,t2F,Wt1cmd,Wtdiffcmd,Whact, ...
                Wcact,Whrate,Wcrate,Wt1perf,Wt2perf,Wt1noise,Wt2noise, ...
                   sum1,sum2,sum3,sum4,sum5,inputs,outputs);

disp('Weighted open-loop model: ')
P
```

```
Weighted open-loop model:

P =

  Uncertain continuous-time state-space model with 10 outputs, 6 inputs, 18 states.
  The model uncertainty consists of the following blocks:
    delta1: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    delta2: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    delta3: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences

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

H-infinity Controller Design

By constructing the weights and weighted open loop of Figure 8, we have recast the control problem as a closed-loop gain minimization. Now we can easily compute a gain-minimizing control law for the nominal tank models:

```
nmeas = 4;        % Number of measurements
nctrls = 2;        % Number of controls
[k0,g0,gamma0] = hinfsyn(P.NominalValue,nmeas,nctrls);
gamma0
```

```
gamma0 =

    0.9016
```

The smallest achievable closed-loop gain is about 0.9, which shows us that our frequency-domain tracking performance specifications are met by the controller `k0`. Simulating this design in the time domain is a reasonable way to check that we have correctly set the performance weights. First, we create a closed-loop model mapping the input signals [`t1ref`; `t2ref`; `t1noise`; `t2noise`] to the output signals [`h1`; `t1`; `t2`; `fhc`; `fcc`]:

```
inputs = {'t1ref', 't2ref', 't1noise', 't2noise', 'fhc', 'fcc'};
outputs = {'y_tank1', 'y_tank2', 'fhc', 'fcc', 'y_t1ref', 'y_t2ref', ...
                'y_t1Fn', 'y_t2Fn'};

hot_act(1).InputName = 'fhc'; hot_act(1).OutputName = 'y_hot_act';
cold_act(1).InputName = 'fcc'; cold_act(1).OutputName = 'y_cold_act';

tank1nom.InputName = [hot_act(1).OutputName cold_act(1).OutputName];
tank1nom.OutputName = 'y_tank1';
tank2nom.InputName = tank1nom.OutputName;
tank2nom.OutputName = 'y_tank2';

t1F.InputName = tank1nom.OutputName(2); t1F.OutputName = 'y_t1F';
t2F.InputName = tank2nom.OutputName; t2F.OutputName = 'y_t2F';

I_tref = zpk(eye(2));
I_tref.InputName = {'t1ref', 't2ref'}; I_tref.OutputName = {'y_t1ref', 'y_t2ref'};

sum1 = sumblk('y_t1Fn = y_t1F + t1noise');
sum2 = sumblk('y_t2Fn = y_t2F + t2noise');

simlft = connect(tank1nom,tank2nom,hot_act(1),cold_act(1),t1F,t2F,I_tref,sum1,sum2,inputs,outputs
% Close the loop with the H-infinity controller |k0|
sim_k0 = lft(simlft,k0);
sim_k0.InputName = {'t1ref'; 't2ref'; 't1noise'; 't2noise'};
sim_k0.OutputName = {'h1'; 't1'; 't2'; 'fhc'; 'fcc'};
```

Now we simulate the closed-loop response when ramping down the setpoints for `t1` and `t2` between 80 seconds and 100 seconds:

```
time=0:800;
t1ref = (time>=80 & time<100).*(time-80)*-0.18/20 + ...
    (time>=100)*-0.18;
t2ref = (time>=80 & time<100).*(time-80)*-0.2/20 + ...
    (time>=100)*-0.2;
t1noise = Wt1noise.k * randn(size(time));
t2noise = Wt2noise.k * randn(size(time));

y = lsim(sim_k0,[t1ref ; t2ref ; t1noise ; t2noise],time);
```

Next, we add the simulated outputs to their steady state values and plot the responses:

```
h1 = h1ss+y(:,1);
t1 = t1ss+y(:,2);
t2 = t2ss+y(:,3);
fhc = fhss/fs+y(:,4); % Note scaling to actuator
fcc = fcss/fs+y(:,5); % Limits (0<= fhc <= 1) etc.
```

In this code, we plot the outputs, `t1` and `t2`, as well as the height `h1` of tank 1:

```
plot(time,h1,'--',time,t1,'-',time,t2,'-.');
xlabel('Time (sec)')
ylabel('Measurements')
title('Step Response of H-infinity Controller k0')
legend('h1','t1','t2');
grid
```

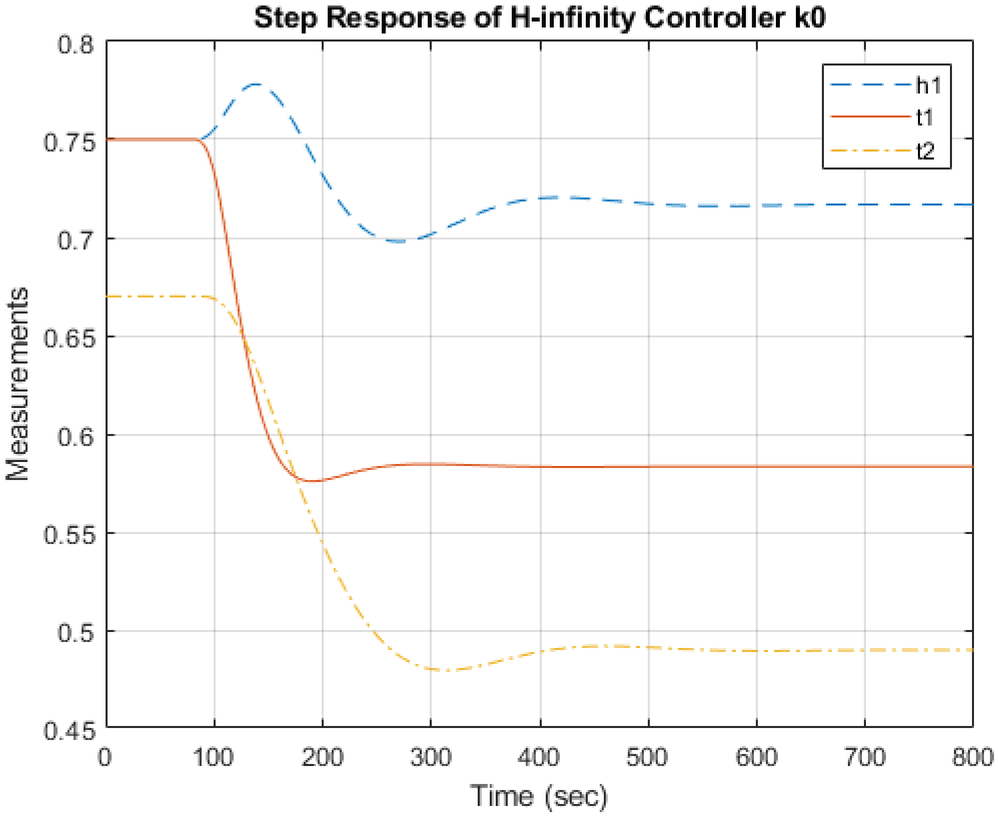


Figure 10: Step response of H-infinity controller k0.

Next we plot the commands to the hot and cold actuators.

```
plot(time,fhc,'-',time,fcc,'-.');
xlabel('Time: seconds')
ylabel('Actuators')
title('Actuator Commands for H-infinity Controller k0')
legend('fhc','fcc');
grid
```

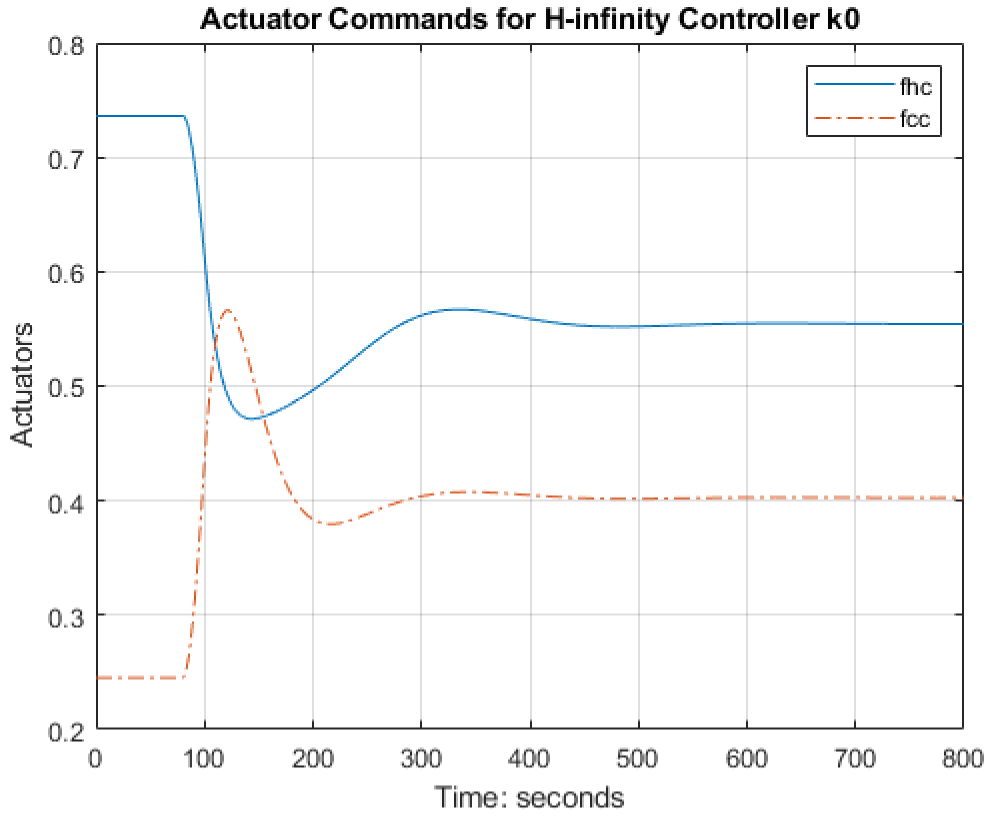


Figure 11: Actuator commands for H-infinity controller k0.

Robustness of the H-infinity Controller

The H-infinity controller `k0` is designed for the nominal tank models. Let's look at how well its fares for perturbed model within the model uncertainty bounds. We can compare the nominal closed-loop performance `gamma0` with the worst-case performance over the model uncertainty set. (see "Uncertainty on Model Dynamics" for more information.)

```
clpk0 = lft(P,k0);

% Compute and plot worst-case gain
wcsigmaplot(clpk0,{1e-4,1e2})
ylim([-20 10])
```

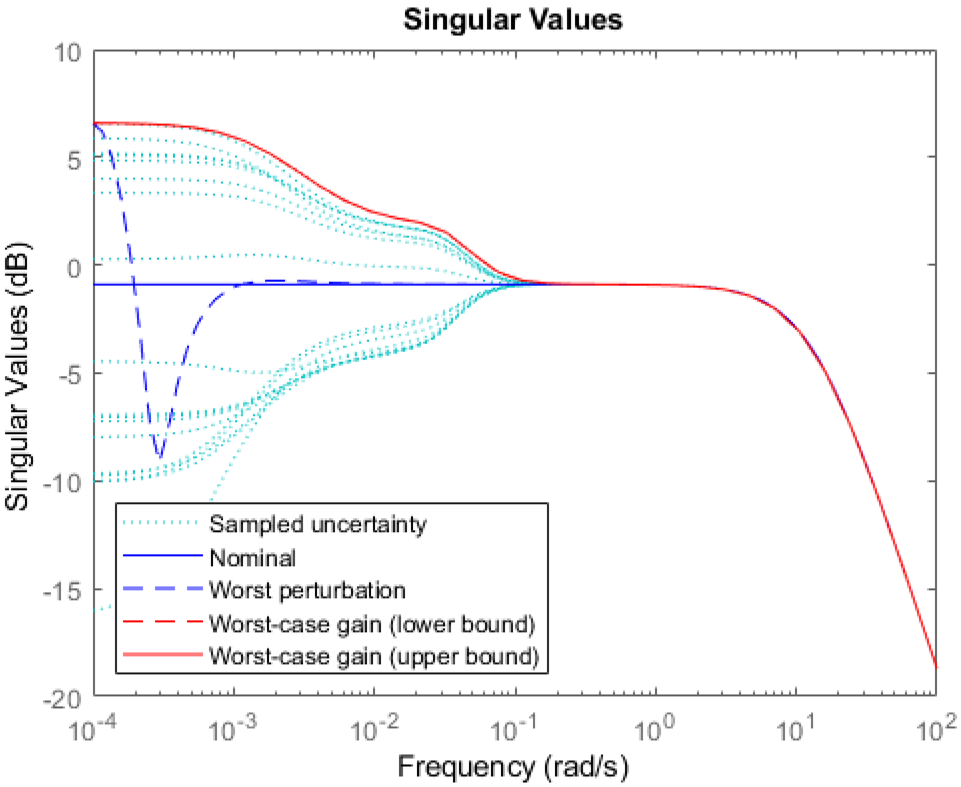


Figure 12: Performance analysis for controller k0.

The worst-case performance of the closed-loop is significantly worse than the nominal performance which tells us that the H-infinity controller `k0` is not robust to modeling errors.

Mu Controller Synthesis

To remedy this lack of robustness, we will use `musyn` to design a controller that takes into account modeling uncertainty and delivers consistent performance for the nominal and perturbed models.

```
[kmu,bnd] = musyn(P,nmeas,nctrls);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step       Peak MU       D Fit             D
    1           8.814         2.862        2.888             4
    2           2.495         2.088        2.109            10
    3           1.355         1.332         1.34            10
    4           1.201         1.199        1.209            22
    5           1.194         1.191        1.198            20
    6           1.192          1.19        1.193            20

Best achieved robust performance: 1.19
```

As before, we can simulate the closed-loop responses with the controller kmu

```
sim_kmu = lft(simlft,kmu);
y = lsim(sim_kmu,[t1ref;t2ref;t1noise;t2noise],time);
h1 = h1ss+y(:,1);
t1 = t1ss+y(:,2);
t2 = t2ss+y(:,3);
fhc = fhss/fs+y(:,4); % Note scaling to actuator
fcc = fcss/fs+y(:,5); % Limits (0<= fhc <= 1) etc.

% Plot |t1| and |t2| as well as the height |h1| of tank 1
plot(time,h1,'--',time,t1,'-',time,t2,'-.');
xlabel('Time: seconds')
ylabel('Measurements')
title('Step Response of mu Controller kmu')
legend('h1','t1','t2');
grid
```

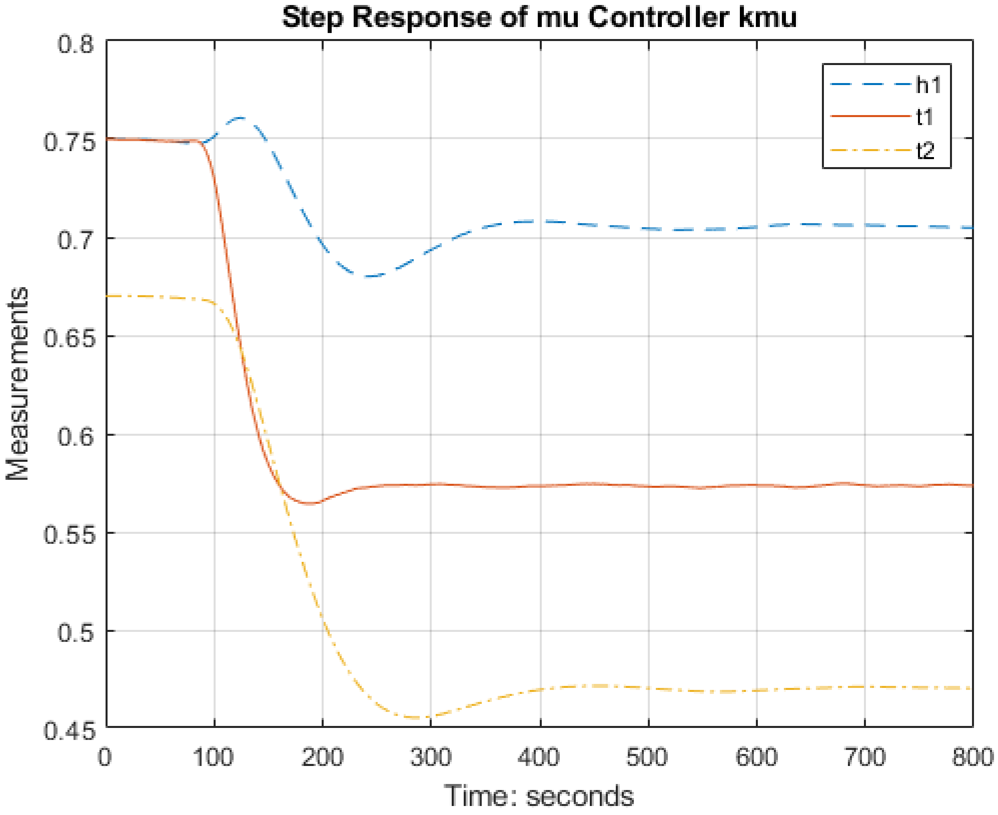


Figure 13: Step response of mu controller kmu.

These time responses are comparable with those for k0, and show only a slight performance degradation. However, kmu fares better regarding robustness to unmodeled dynamics.

```
% Worst-case performance for kmu
clpmu = lft(P,kmu);
wcsigmaplot(clpmu,{1e-4,1e2})
ylim([-20 10])
```

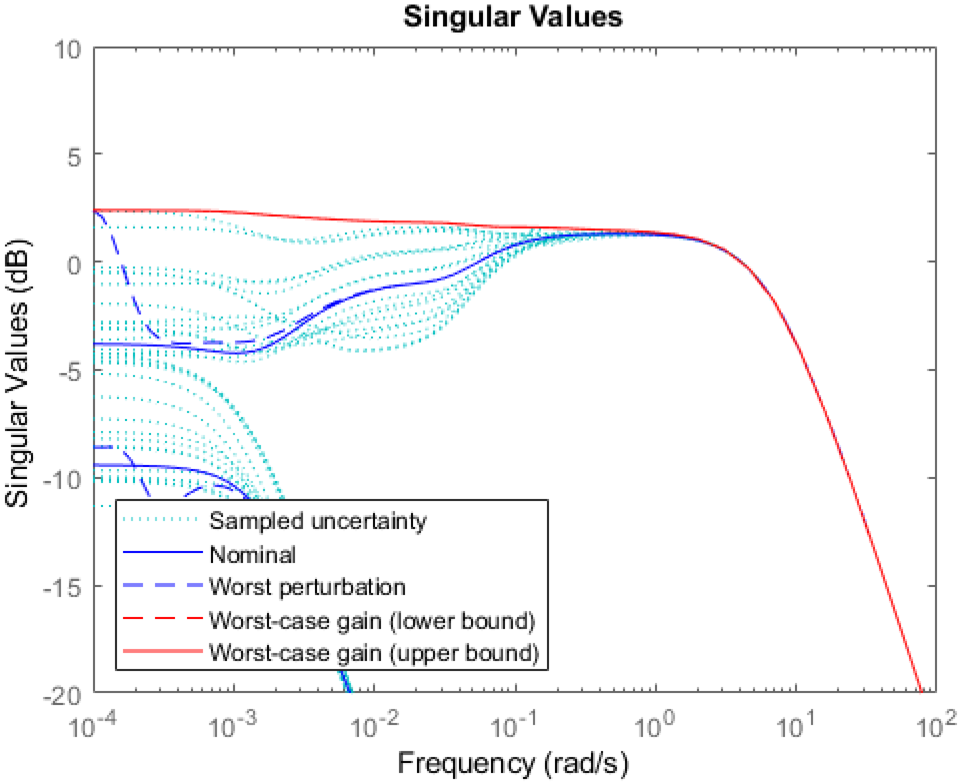


Figure 14: Performance analysis for controller kmu.

You can use `wcgain` to directly compute the worst-case gain across frequency (worst-case peak gain or worst-case H-infinity norm). You can also compute its sensitivity to each uncertain element. Results show that the worst-case peak gain is most sensitive to changes in the range of `delta2`.

```
opt = wcOptions('Sensitivity','on');
[wcg,wcu,wcinfo] = wcgain(clpmu,opt);
wcg
```

```
wcg = 

  struct with fields:

           LowerBound: 1.3165
           UpperBound: 1.3192
    CriticalFrequency: 1.9799e-11
```

```
wcinfo.Sensitivity
```

```
ans = 

  struct with fields:

    delta1: 0
    delta2: 60
    delta3: 10
```

See Also

`hinfsyn` | `musyn` | `wcgain`

More About

- “H-Infinity Performance”
- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “Simultaneous Stabilization Using Robust Control” on page 3-54

Simultaneous Stabilization Using Robust Control

This example uses the Robust Control Toolbox™ commands ucover and musyn to design a high-performance controller for a family of unstable plants.

Plant Uncertainty

The nominal plant model consists of a first-order unstable system.

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

The family of perturbed plants are variations of Pnom. All plants have a single unstable pole but the location of this pole varies across the family.

```
p1 = Pnom*tf(1,[.06 1]);             % extra lag
p2 = Pnom*tf([-.02 1],[.02 1]);      % time delay
p3 = Pnom*tf(50^2,[1 2*.1*50 50^2]);  % high frequency resonance
p4 = Pnom*tf(70^2,[1 2*.2*70 70^2]);  % high frequency resonance
p5 = tf(2.4,[1 -2.2]);               % pole/gain migration
p6 = tf(1.6,[1 -1.8]);               % pole/gain migration
```

Covering the Uncertain Model

For feedback design purposes, we need to replace this set of models with a single uncertain plant model whose range of behaviors includes p1 through p6. This is one use of the command ucover. This command takes an array of LTI models Parray and a nominal model Pnom and models the difference Parray-Pnom as multiplicative uncertainty in the system dynamics.

Because ucover expects an array of models, use the stack command to gather the plant models p1 through p6 into one array.

```
Parray = stack(1,p1,p2,p3,p4,p5,p6);
```

Next, use ucover to "cover" the range of behaviors Parray with an uncertain model of the form

```
P = Pnom * (1 + Wt * Delta)
```

where all uncertainty is concentrated in the "unmodeled dynamics" Delta (a ultidyn object). Because the gain of Delta is uniformly bounded by 1 at all frequencies, a "shaping" filter Wt is used to capture how the relative amount of uncertainty varies with frequency. This filter is also referred to as the uncertainty weighting function. Try a 4th-order filter Wt for this example:

```
orderWt = 4;
Parrayg = frd(Parray,logspace(-1,3,60));
[P,Info] = ucover(Parrayg,Pnom,orderWt,'InputMult');
```

The resulting model P is a single-input, single-output uncertain state-space (USS) object with nominal value Pnom.

```
P
```

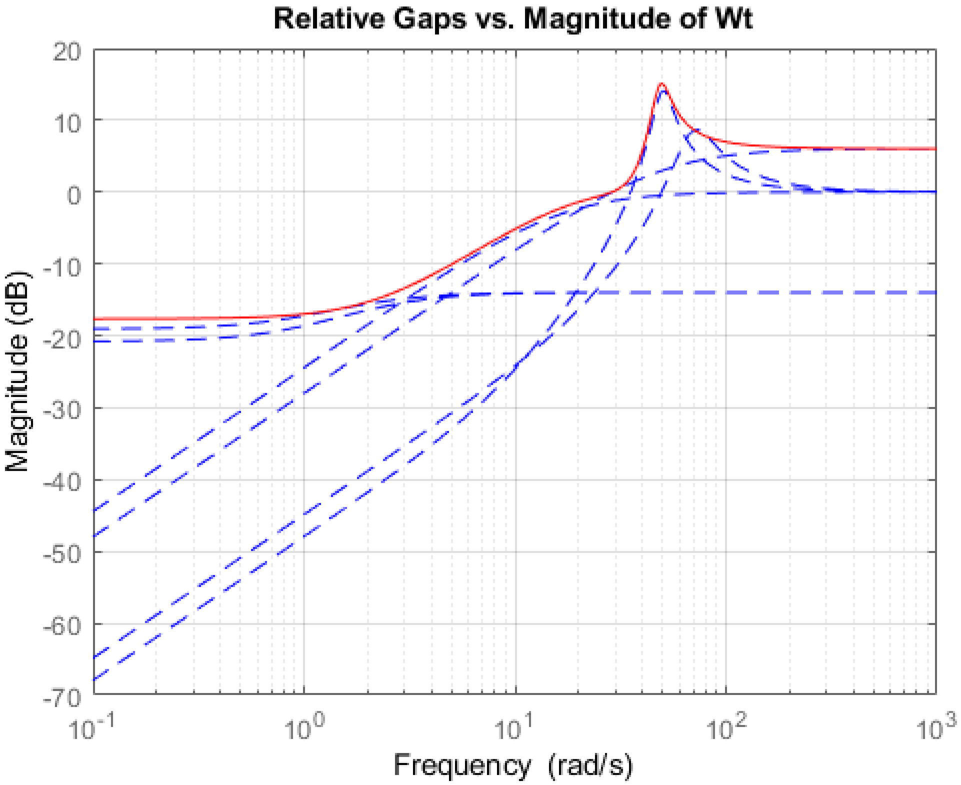
```
P =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 5 states.
  The model uncertainty consists of the following blocks:
    Parrayg_InputMultDelta: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences

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

```
tf(P.NominalValue)
```

```
ans =
 
    2
  -----
  s - 2
 
Continuous-time transfer function.
```

A Bode magnitude plot confirms that the shaping filter Wt "covers" the relative variation in plant behavior. As a function of frequency, the uncertainty level is 30% at 5 rad/sec (-10dB = 0.3) , 50% at 10 rad/sec, and 100% beyond 29 rad/sec.

```
Wt = Info.W1;
bodemag((Pnom-Parray)/Pnom,'b--',Wt,'r'); grid
title('Relative Gaps vs. Magnitude of Wt')
```



Creating the Open-loop Design Model

To design a robust controller for the uncertain plant model P, we choose a desired closed-loop bandwidth and minimize the sensitivity to disturbances at the plant output. The control structure is shown below. The signals d and n are the load disturbance and measurement noise. The controller uses a noisy measurement of the plant output y to generate the control signal u.

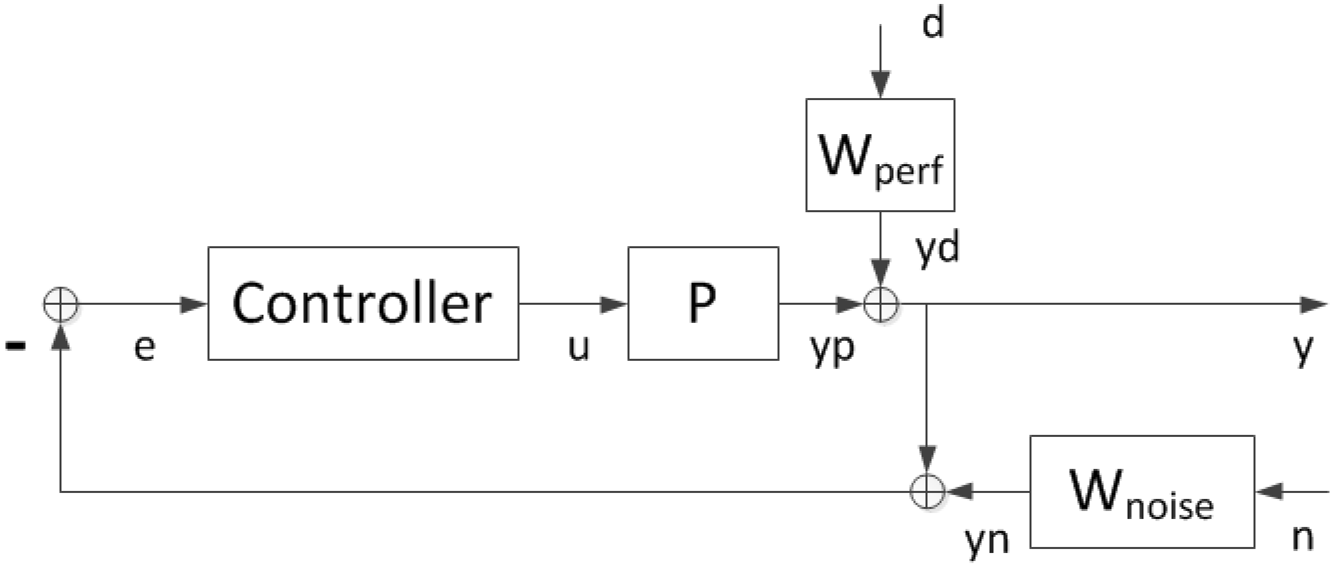


Figure 1: Control Structure.

The filters Wperf and Wnoise are selected to enforce the desired bandwidth and some adequate roll-off. The closed-loop transfer function from [d;n] to y is

```
    y = [Wperf * S , Wnoise * T] [d;n]
```

where S=1/(1+PC) and T=PC/(1+PC) are the sensitivity and complementary sensitivity functions. If we design a controller that keeps the closed-loop gain from [d;n] to y below 1, then

```
    |S| < 1/|Wperf| ,    |T| < 1/|Wnoise|
```

By choosing appropriate magnitude profiles for Wperf and Wnoise, we can enforce small sensitivity (S) inside the bandwidth and adequate roll-off (T) outside the bandwidth.

For example, choose Wperf as a first-order low-pass filter with a DC gain of 500 and a gain crossover at the desired bandwidth desBW:

```
desBW = 4.5;
Wperf = makeweight(500,desBW,0.33);
tf(Wperf)
```

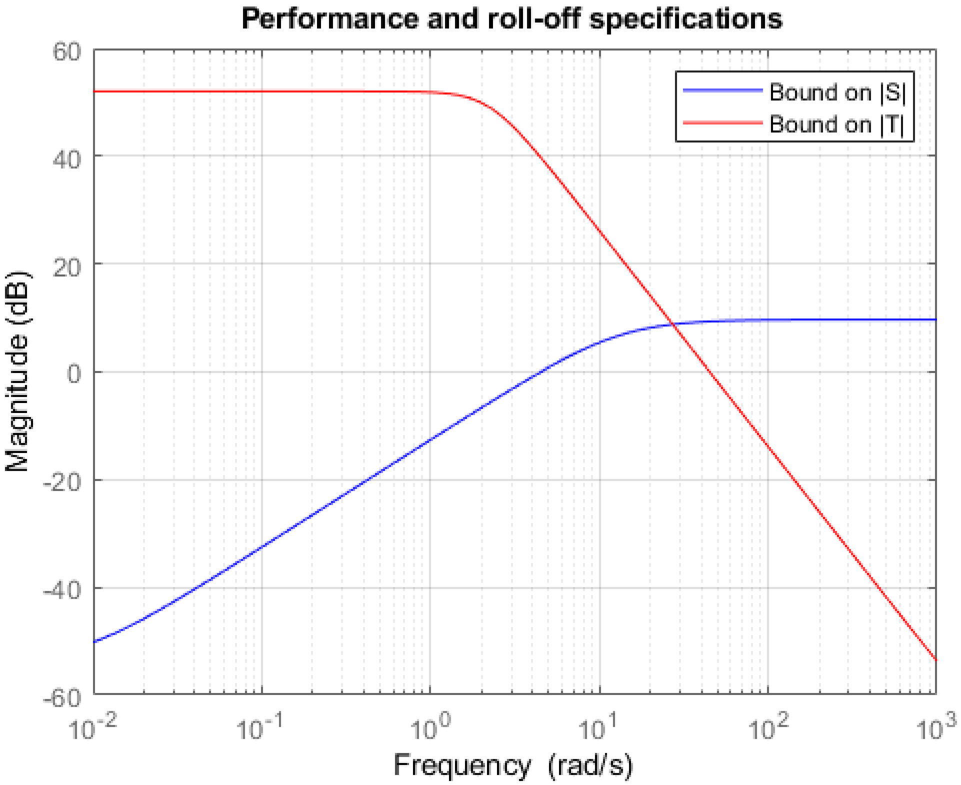
```
ans =
 
  0.33 s + 4.248
  --------------
   s + 0.008496
 
Continuous-time transfer function.
```

Similarly, pick Wnoise as a second-order high-pass filter with a magnitude of 1 at 10*desBW. This will force the open-loop gain PC to roll-off with a slope of -2 for frequencies beyond 10*desBW.

```
NF = (10*desBW)/20;  % numerator corner frequency
DF = (10*desBW)*50;  % denominator corner frequency
Wnoise = tf([1/NF^2  2*0.707/NF  1],[1/DF^2  2*0.707/DF  1]);
Wnoise = Wnoise/abs(freqresp(Wnoise,10*desBW))
Wnoise =

    0.1975 s^2 + 0.6284 s + 1
  ------------------------------
  7.901e-05 s^2 + 0.2514 s + 400

Continuous-time transfer function.
```

Verify that the bounds `1/Wperf` and `1/Wnoise` on `S` and `T` do enforce the desired bandwidth and roll-off.

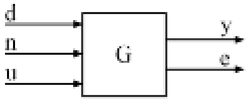
```
bodemag(1/Wperf,'b',1/Wnoise,'r',{1e-2,1e3}), grid
title('Performance and roll-off specifications')
legend('Bound on |S|','Bound on |T|','Location','NorthEast')
```



Next use `connect` to build the open-loop interconnection (block diagram in Figure 1 without the controller block). Specify each block appearing in Figure 1, name the signals coming in and out of each block, and let `connect` do the wiring:

```
P.u = 'u';    P.y = 'yp';
Wperf.u = 'd';   Wperf.y = 'Wperf';
Wnoise.u = 'n';  Wnoise.y = 'Wnoise';
S1 = sumblk('e = -ym');
S2 = sumblk('y = yp + Wperf');
S3 = sumblk('ym = y + Wnoise');
G = connect(P,Wperf,Wnoise,S1,S2,S3,{'d','n','u'},{'y','e'});
```

`G` is a 3-input, 2-output uncertain system suitable for robust controller synthesis with musyn.



Robust Controller Synthesis

The design is carried out with the automated robust design command musyn. The target bandwidth is 4.5 rad/s.

```
ny = 1; nu = 1;
[C,muPerf] = musyn(G,ny,nu);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter         K Step        Peak MU       D Fit             D
    1           353.6          249.5       251.9             0
    2           70.74          9.965       10.05             4
    3           1.981          1.601       1.619             8
    4           1.159          1.159        1.18            10
    5           1.089          1.089       1.098            10
    6           1.046          1.046       1.051             8
    7           1.025          1.025       1.033             8
    8           1.015          1.015       1.022             8
    9           1.013          1.013       1.017             8
   10           1.011          1.011       1.026            10

Best achieved robust performance: 1.01
```

When the robust performance indicator muPerf is near 1, the controller achieves the target closed-loop bandwidth and roll-off. As a rule of thumb, if muPerf is less than 0.85, then the performance can be improved upon, and if muPerf is greater than 1.2, then the desired closed-loop bandwidth is not achievable for the specified plant uncertainty.

Here muPerf is approximately 1 so the objectives are met. The resulting controller C has 18 states:

```
size(C)
```

```
State-space model with 1 outputs, 1 inputs, and 16 states.
```

You can use the reduce and musynperf commands to simplify this controller. Compute approximations of orders 1 through 17.

```
NxC = order(C);
Cappx = reduce(C,1:NxC);
```

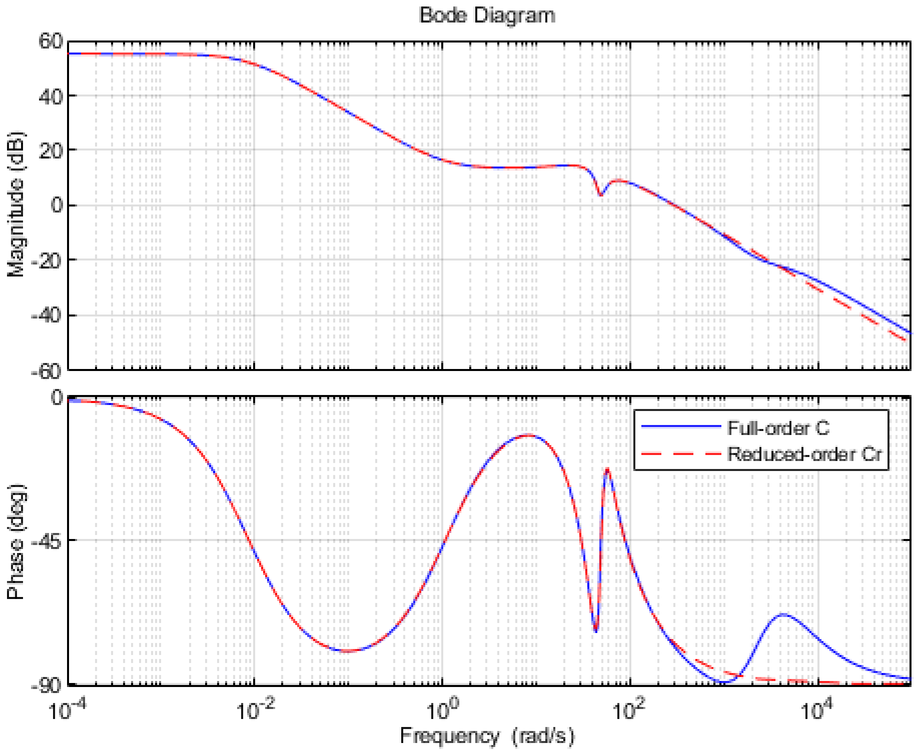
For each reduced-order controller, use musynperf to compute the robust performance indicator and compare it with muPerf. Keep the lowest-order controller with performance no worse than 1.05 * muPerf, a performance degradation of 5% or less.

```
for k=1:NxC
   Cr = Cappx(:,:,k);  % controller of order k
   bnd = musynperf(lft(G,Cr));
   if bnd.UpperBound < 1.05 * muPerf
```

```
      break % abort with the first controller meeting the performance goal
   end
end

order(Cr)

ans = 6
```

This yields a 6th-order controller `Cr` with comparable performance. Compare `Cr` with the full-order controller `C`.

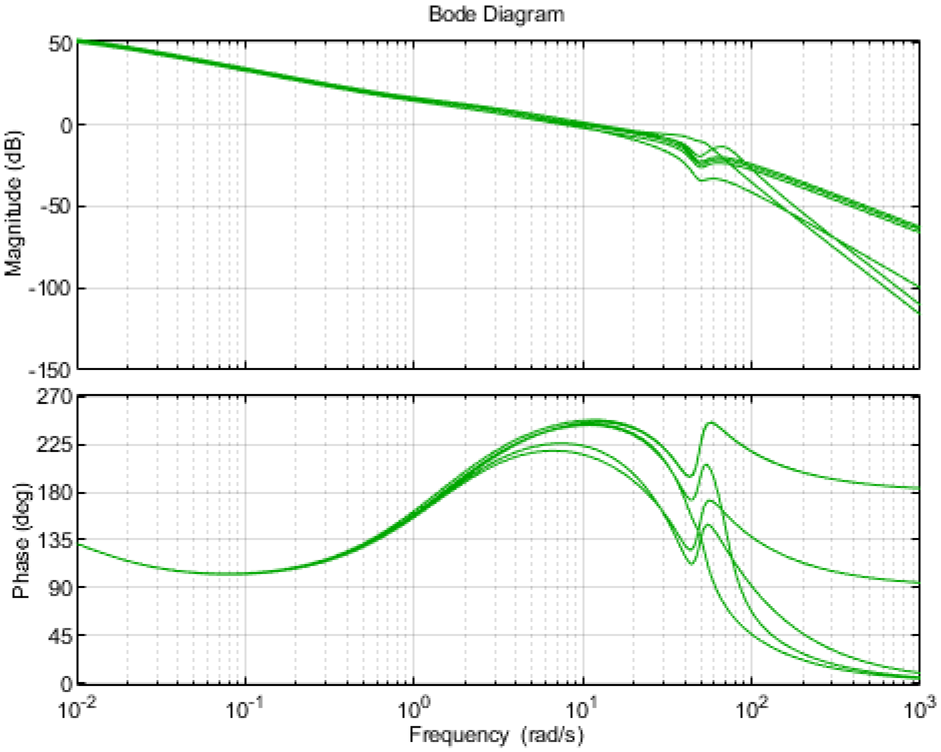
```
opt = bodeoptions;
opt.Grid = 'on';
opt.PhaseMatching = 'on';
bodeplot(C,'b',Cr,'r--',opt)
legend('Full-order C','Reduced-order Cr','Location','NorthEast')
```



Robust Controller Validation

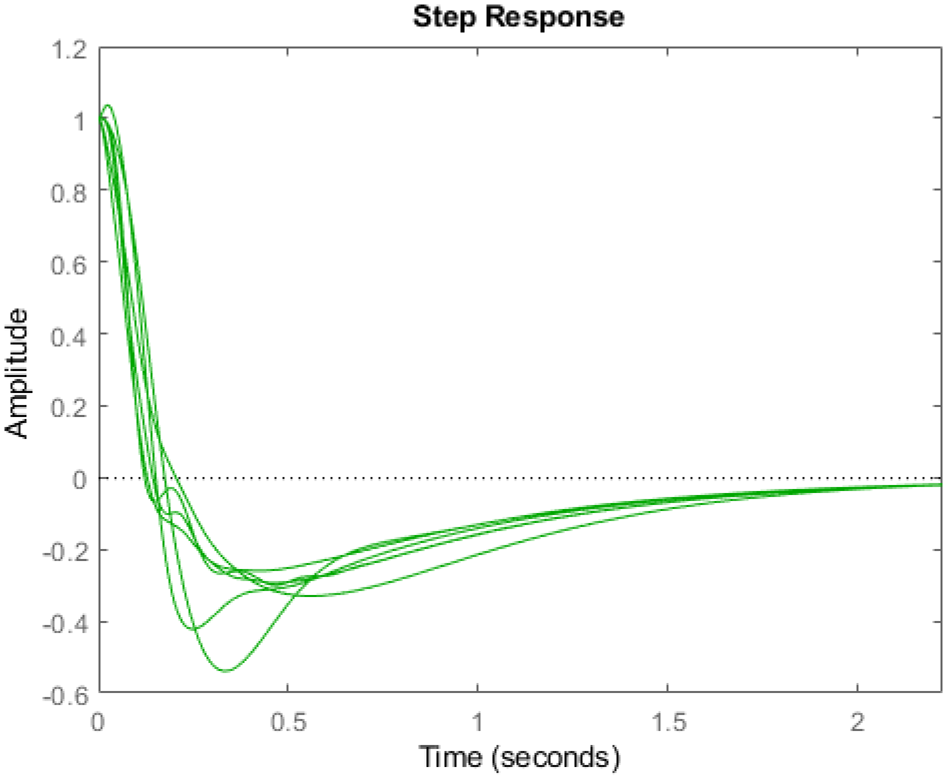
Plot the open-loop responses of the plant models `p1` through `p6` with the simplified controller `Cr`.

```
bodeplot(Parray*Cr,'g',{1e-2,1e3},opt);
```



Plot the responses to a step disturbance at the plant output. These are consistent with the desired closed-loop bandwidth and robust to the plant variations, as expected from a Robust Performance mu-value of approximately 1.

```
step(feedback(1,Parray*Cr),'g',10/desBW);
```



Varying the Target Closed-Loop Bandwidth

The same design process can be repeated for different closed-loop bandwidth values `desBW`. Doing so yields the following results:

- Using `desBW` = 8 yields a good design with robust performance `muPerf` of 1.09. The step responses across the `Parray` family are consistent with a closed-loop bandwidth of 8 rad/s.
- Using `desBW` = 20 yields a poor design with robust performance `muPerf` of 1.35. This is expected because this target bandwidth is in the vicinity of very large plant uncertainty. Some of the step responses for the plants `p1,...,p6` are actually unstable.
- Using `desBW` = 0.3 yields a poor design with robust performance `muPerf` of 2.2. This is expected because `Wnoise` imposes roll-off past 3 rad/s, which is too close to the natural frequency of the unstable pole (2 rad/s). In other words, proper control of the unstable dynamics requires a higher bandwidth than specified.

See Also

`makeweight` | `musyn` | `ucover`

More About

- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “Control of Aircraft Lateral Axis Using Mu Synthesis” on page 3-62

Control of Aircraft Lateral Axis Using Mu Synthesis

This example shows how to use mu-analysis and synthesis tools in the Robust Control Toolbox™. It describes the design of a robust controller for the lateral-directional axis of an aircraft during powered approach to landing. The linearized model of the aircraft is obtained for an angle-of-attack of 10.5 degrees and airspeed of 140 knots.

Performance Specifications

The illustration below shows a block diagram of the closed-loop system. The diagram includes the nominal aircraft model, the controller K, as well as elements capturing the model uncertainty and performance objectives (see next sections for details).

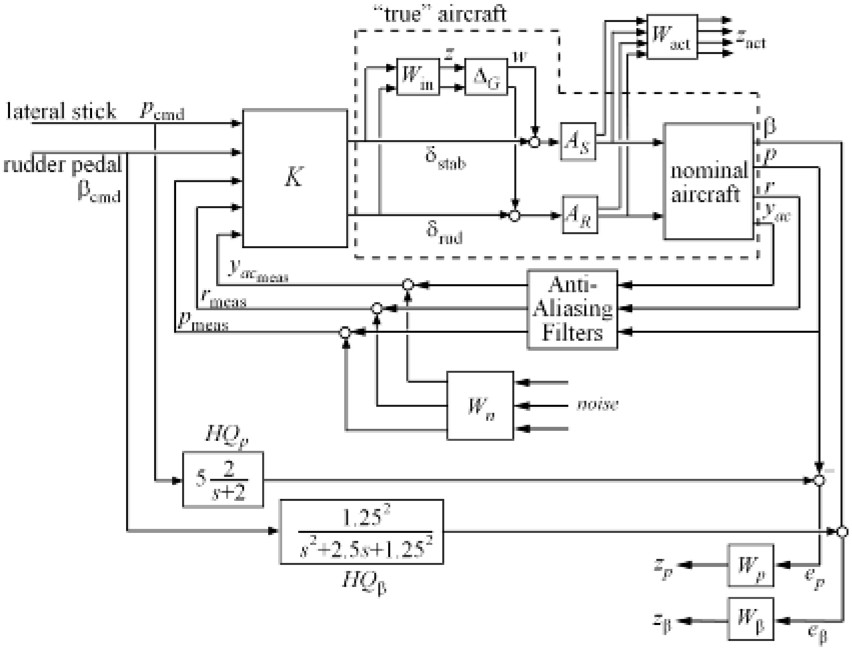


Figure 1: Robust Control Design for Aircraft Lateral Axis

The design goal is to make the airplane respond effectively to the pilot's lateral stick and rudder pedal inputs. The performance specifications include:

- Decoupled responses from lateral stick p_cmd to roll rate p and from rudder pedals beta_cmd to side-slip angle beta. The lateral stick and rudder pedals have a maximum deflection of +/- 1 inch.
- The aircraft handling quality (HQ) response from lateral stick to roll rate p should match the first-order response.

```
HQ_p    = 5.0 * tf(2.0,[1 2.0]);
step(HQ_p), title('Desired response from lateral stick to roll rate (Handling Quality)')
```

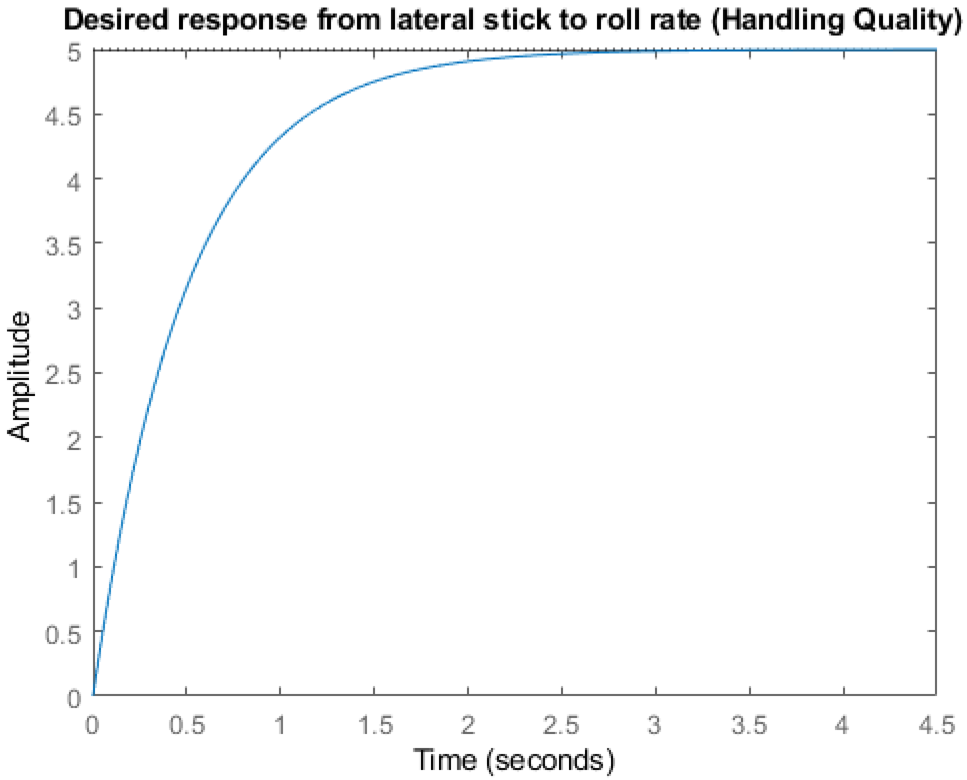


Figure 2: Desired response from lateral stick to roll rate.

- The aircraft handling quality response from the rudder pedals to the side-slip angle `beta` should match the damped second-order response.

```
HQ_beta = -2.5 * tf(1.25^2,[1 2.5 1.25^2]);
step(HQ_beta), title('Desired response from rudder pedal to side-slip angle (Handling Quality)')
```

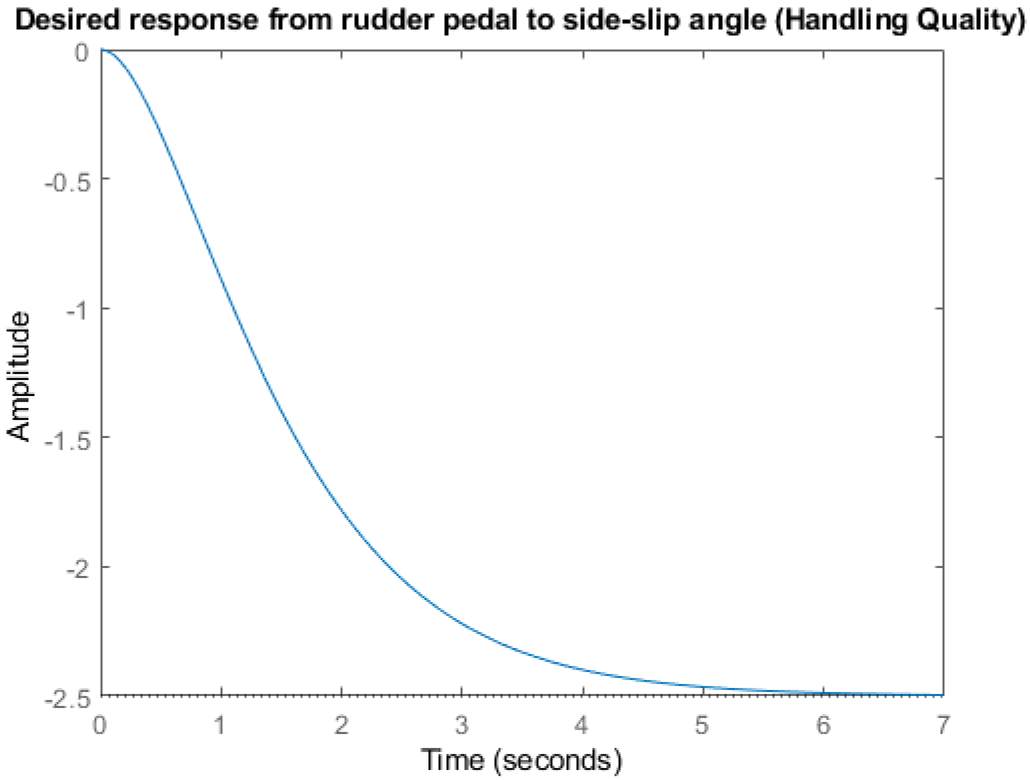


Figure 3: Desired response from rudder pedal to side-slip angle.

- The stabilizer actuators have +/- 20 deg and +/- 50 deg/s limits on their deflection angle and deflection rate. The rudder actuators have +/- 30 deg and +/-60 deg/s deflection angle and rate limits.
- The three measurement signals (roll rate `p`, yaw rate `r`, and lateral acceleration `yac`) are filtered through second-order anti-aliasing filters:

```
freq = 12.5 * (2*pi);  % 12.5 Hz
zeta = 0.5;
yaw_filt = tf(freq^2,[1 2*zeta*freq freq^2]);
lat_filt = tf(freq^2,[1 2*zeta*freq freq^2]);

freq = 4.1 * (2*pi);  % 4.1 Hz
zeta = 0.7;
roll_filt = tf(freq^2,[1 2*zeta*freq freq^2]);

AAFilters = append(roll_filt,yaw_filt,lat_filt);
```

From Specs to Weighting Functions

H-infinity design algorithms seek to minimize the largest closed-loop gain across frequency (H-infinity norm). To apply these tools, we must first recast the design specifications as constraints on the closed-loop gains. We use **weighting functions** to "normalize" the specifications across frequency and to equally weight each requirement.

We can express the design specs in terms of weighting functions as follows:

- To capture the limits on the actuator deflection magnitude and rate, pick a diagonal, constant weight `W_act`, corresponding to the stabilizer and rudder deflection rate and deflection angle limits.

```
W_act = ss(diag([1/50,1/20,1/60,1/30]));
```

- Use a 3x3 diagonal, high-pass filter `W_n` to model the frequency content of the sensor noise in the roll rate, yaw rate, and lateral acceleration channels.

```
W_n = append(0.025,tf(0.0125*[1 1],[1 100]),0.025);
clf, bodemag(W_n(2,2)), title('Sensor noise power as a function of frequency')
```

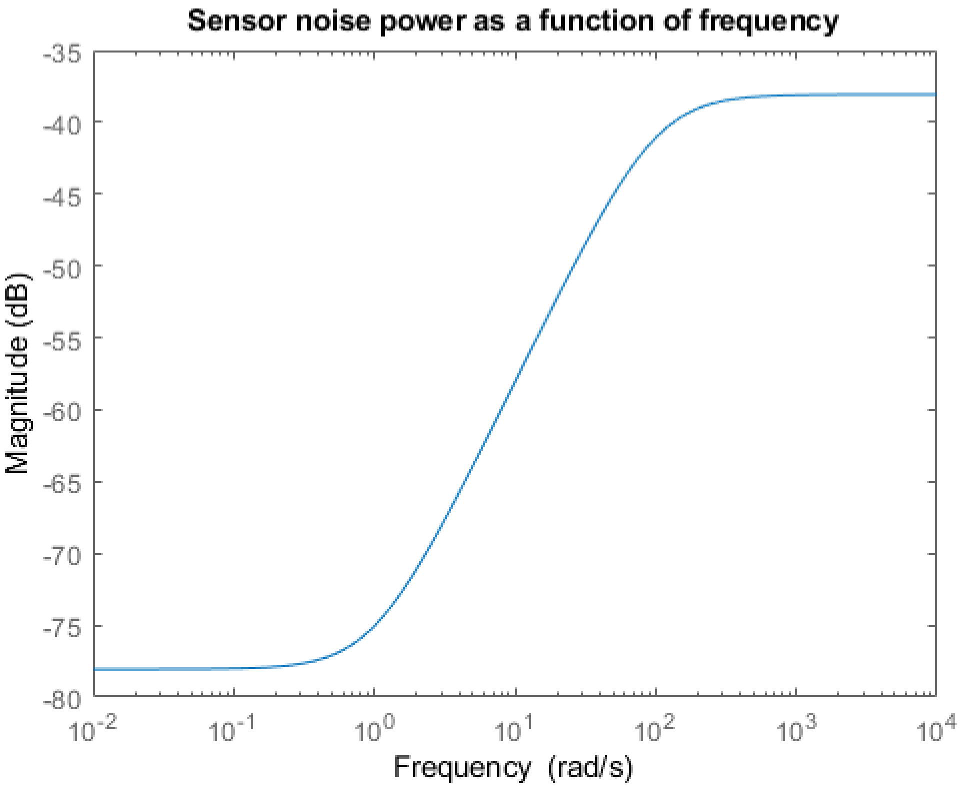


Figure 4: Sensor noise power as a function of frequency

- The response from lateral stick to `p` and from rudder pedal to `beta` should match the handling quality targets `HQ_p` and `HQ_beta`. This is a model-matching objective: to minimize the difference (peak gain) between the desired and actual closed-loop transfer functions. Performance is limited due to a right-half plane zero in the model at 0.002 rad/s, so accurate tracking of sinusoids below 0.002 rad/s is not possible. Accordingly, we'll weight the first handling quality spec with a bandpass filter `W_p` that emphasizes the frequency range between 0.06 and 30 rad/sec.

```
W_p = tf([0.05 2.9 105.93 6.17 0.16],[1 9.19 30.80 18.83 3.95]);
clf, bodemag(W_p), title('Weight on Handling Quality spec')
```

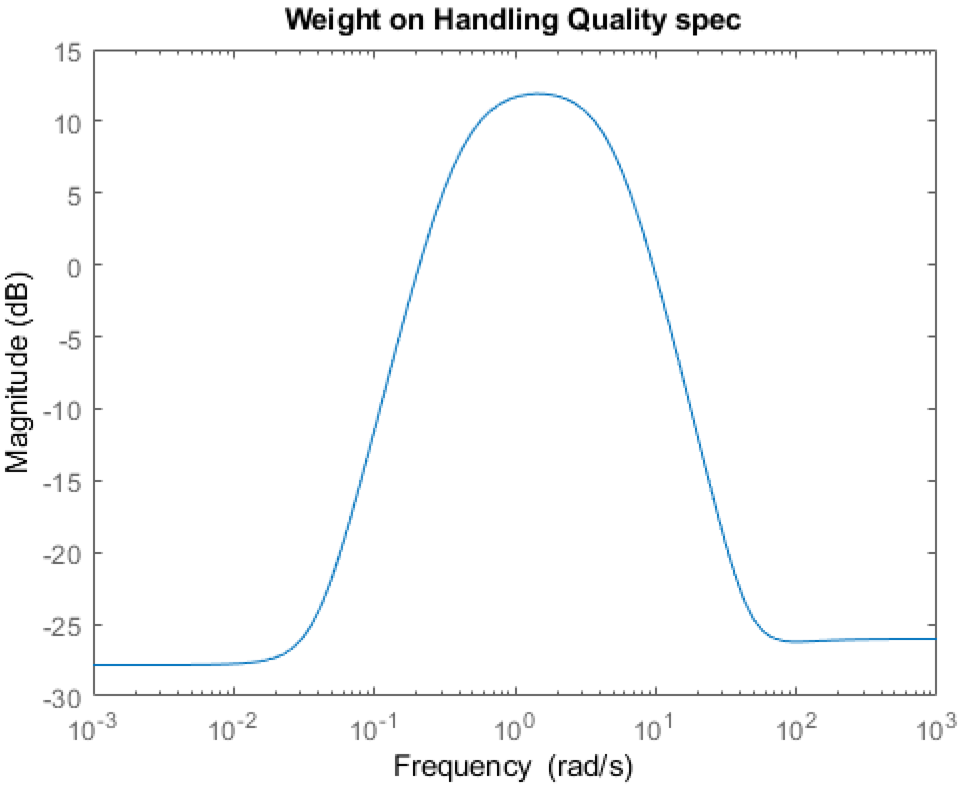


Figure 5: Weight on handling quality spec.

- Similarly, pick `W_beta=2*W_p` for the second handling quality spec

```
W_beta = 2*W_p;
```

Here we scaled the weights `W_act`, `W_n`, `W_p`, and `W_beta` so the closed-loop gain between all external inputs and all weighted outputs is less than 1 at all frequencies.

Nominal Aircraft Model

A pilot can command the lateral-directional response of the aircraft with the lateral stick and rudder pedals. The aircraft has the following characteristics:

- Two control inputs: differential stabilizer deflection `delta_stab` in degrees, and rudder deflection `delta_rud` in degrees.
- Three measured outputs: roll rate `p` in deg/s, yaw rate `r` in deg/s, and lateral acceleration `yac` in g's.
- One calculated output: side-slip angle `beta`.

The nominal lateral directional model `LateralAxis` has four states:

- Lateral velocity `v`
- Yaw rate `r`
- Roll rate `p`

- Roll angle phi

These variables are related by the state space equations:

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

where x = [v; r; p; phi], u = [delta_stab; delta_rud], and y = [beta; p; r; yac].

```
load LateralAxisModel
LateralAxis
```

```
LateralAxis =
 
  A = 
               v         r         p       phi
   v      -0.116    -227.3     43.02     31.63
   r     0.00265    -0.259   -0.1445         0
   p    -0.02114    0.6703    -1.365         0
   phi         0    0.1853         1         0
 
  B = 
        delta_stab   delta_rud
   v        0.0622      0.1013
   r     -0.005252    -0.01121
   p      -0.04666    0.003644
   phi           0           0
 
  C = 
                 v          r          p        phi
   beta     0.2469          0          0          0
   p             0          0       57.3          0
   r             0       57.3          0          0
   yac   -0.002827  -0.007877    0.05106          0
 
  D = 
         delta_stab   delta_rud
   beta           0           0
   p              0           0
   r              0           0
   yac     0.002886    0.002273
 
Continuous-time state-space model.
```

The complete airframe model also includes actuators models A_S and A_R. The actuator outputs are their respective deflection rates and angles. The actuator rates are used to penalize the actuation effort.

```
A_S = [tf([25 0],[1 25]); tf(25,[1 25])];
A_S.OutputName = {'stab_rate','stab_angle'};

A_R = A_S;
A_R.OutputName = {'rud_rate','rud_angle'};
```

Accounting for Modeling Errors

The nominal model only approximates true airplane behavior. To account for unmodeled dynamics, you can introduce a relative term or multiplicative uncertainty W_in*Delta_G at the plant input, where the error dynamics Delta_G have gain less than 1 across frequencies, and the weighting function W_in reflects the frequency ranges in which the model is more or less accurate. There are typically more modeling errors at high frequencies so W_in is high pass.

```
% Normalized error dynamics
Delta_G = ultidyn('Delta_G',[2 2],'Bound',1.0);

% Frequency shaping of error dynamics
w_1 = tf(2.0*[1 4],[1 160]);
w_2 = tf(1.5*[1 20],[1 200]);
W_in = append(w_1,w_2);

bodemag(w_1,'-',w_2,'--')
title('Relative error on nominal model as a function of frequency')
legend('stabilizer','rudder','Location','NorthWest');
```

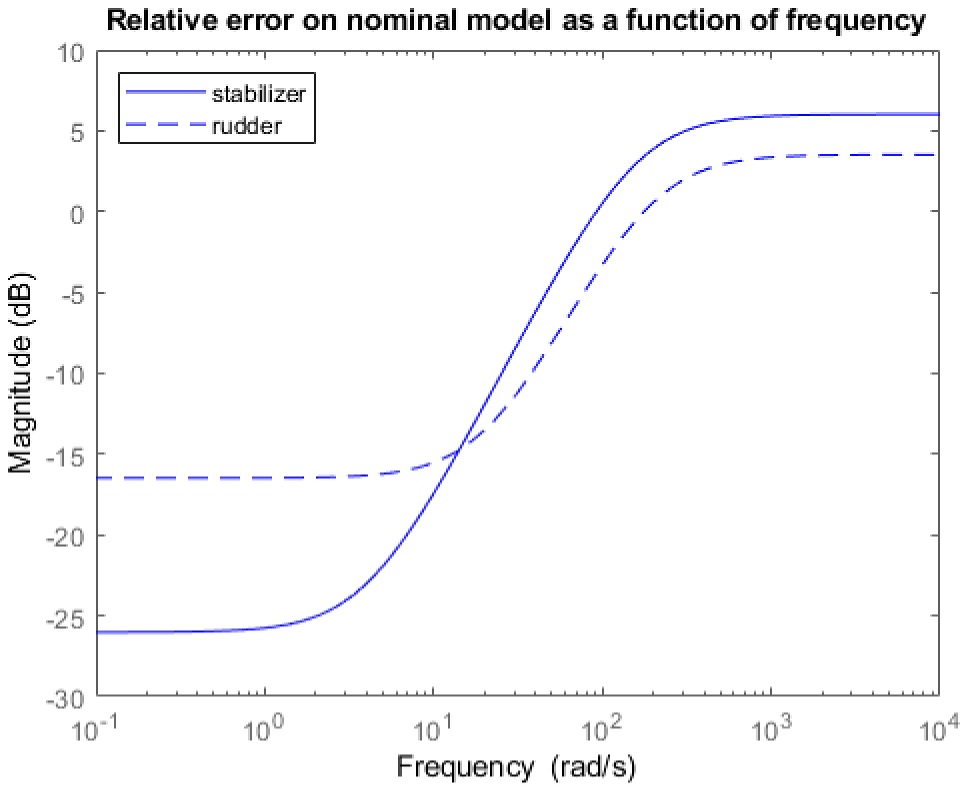


Figure 6: Relative error on nominal aircraft model as a function of frequency.

Building an Uncertain Model of the Aircraft Dynamics

Now that we have quantified modeling errors, we can build an uncertain model of the aircraft dynamics corresponding to the dashed box in the Figure 7 (same as Figure 1):

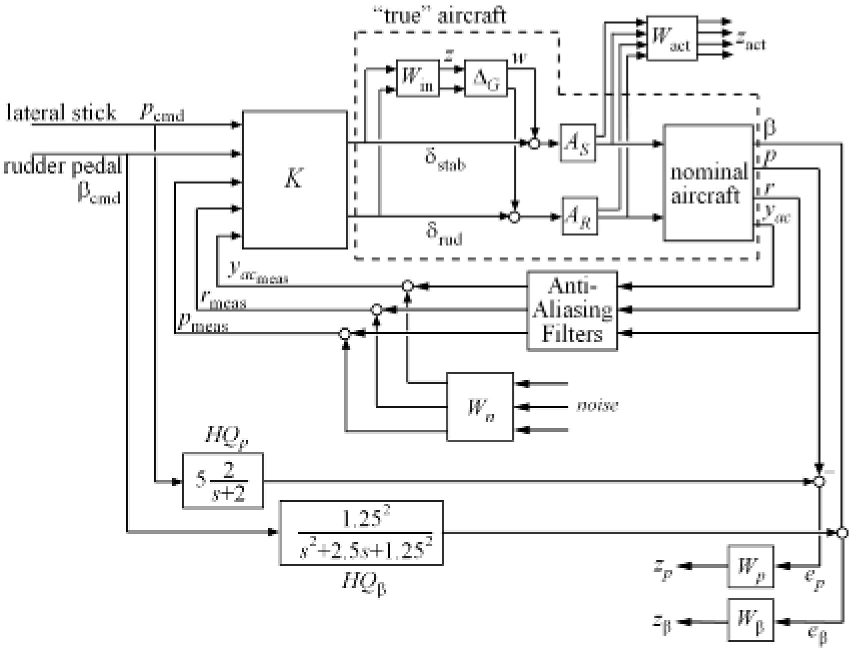


Figure 7: Aircraft dynamics.

Use the `connect` function to combine the nominal airframe model `LateralAxis`, the actuator models `A_S` and `A_R`, and the modeling error description `W_in*Delta_G` into a single uncertain model `Plant_unc` mapping `[delta_stab; delta_rud]` to the actuator and plant outputs:

```
% Actuator model with modeling uncertainty
Act_unc = append(A_S,A_R) * (eye(2) + W_in*Delta_G);
Act_unc.InputName = {'delta_stab','delta_rud'};

% Nominal aircraft dynamics
Plant_nom = LateralAxis;
Plant_nom.InputName = {'stab_angle','rud_angle'};

% Connect the two subsystems
Inputs = {'delta_stab','delta_rud'};
Outputs = [A_S.y ; A_R.y ; Plant_nom.y];
Plant_unc = connect(Plant_nom,Act_unc,Inputs,Outputs);
```

This produces an uncertain state-space (USS) model `Plant_unc` of the aircraft:

```
Plant_unc
```

```
Plant_unc =

  Uncertain continuous-time state-space model with 8 outputs, 2 inputs, 8 states.
  The model uncertainty consists of the following blocks:
    Delta_G: Uncertain 2x2 LTI, peak gain = 1, 1 occurrences

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

Analyzing How Modeling Errors Affect Open-Loop Responses

We can analyze the effect of modeling uncertainty by picking random samples of the unmodeled dynamics `Delta_G` and plotting the nominal and perturbed time responses (Monte Carlo analysis). For example, for the differential stabilizer channel, the uncertainty weight `w_1` implies a 5% modeling error at low frequency, increasing to 100% after 93 rad/sec, as confirmed by the Bode diagram below.

```
% Pick 10 random samples
Plant_unc_sampl = usample(Plant_unc,10);

% Look at response from differential stabilizer to beta
figure('Position',[100,100,560,500])
subplot(211), step(Plant_unc.Nominal(5,1),'r+',Plant_unc_sampl(5,1),'b-',10)
legend('Nominal','Perturbed')

subplot(212), bodemag(Plant_unc.Nominal(5,1),'r+',Plant_unc_sampl(5,1),'b-',{0.001,1e3})
legend('Nominal','Perturbed')
```

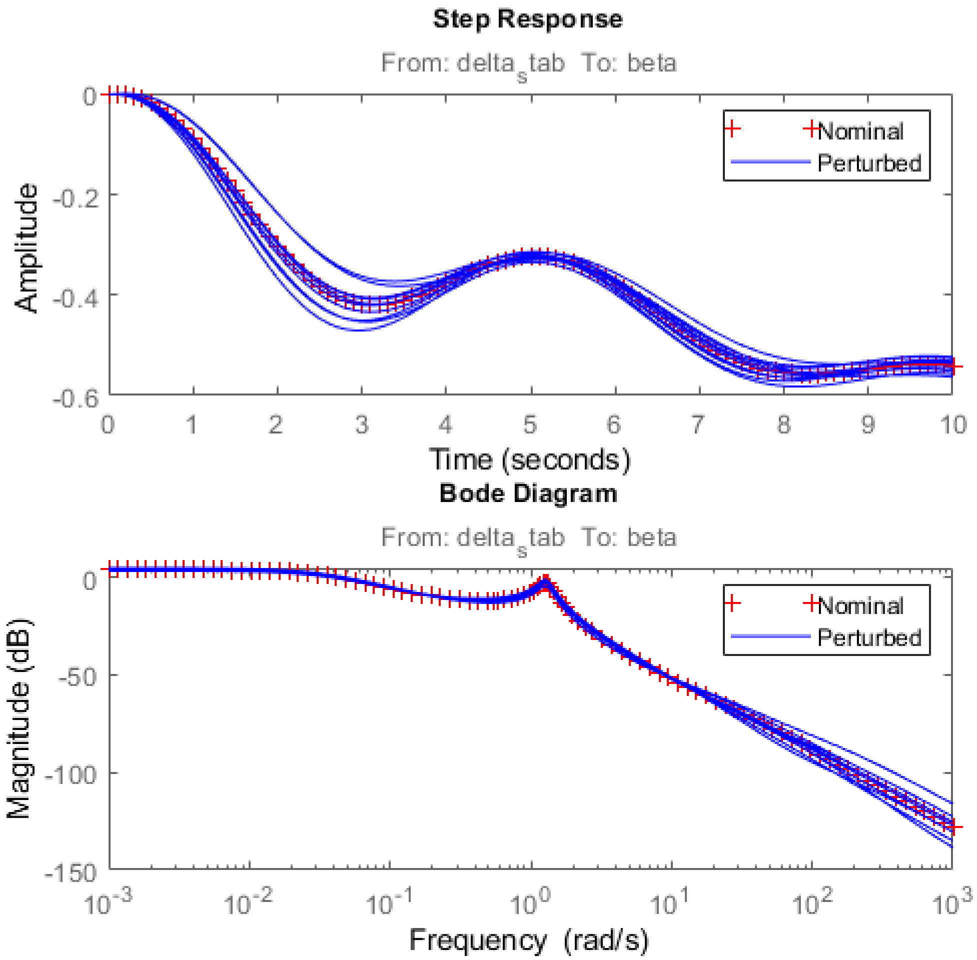


Figure 8: Step response and Bode diagram.

Designing the Lateral-Axis Controller

Proceed with designing a controller that robustly achieves the specifications, where robustly means for any perturbed aircraft model consistent with the modeling error bounds `W_in`.

First we build an open-loop model `OLIC` mapping the external input signals to the performance-related outputs as shown below.

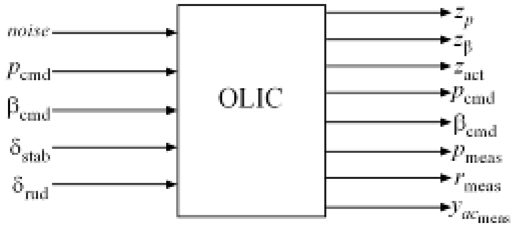


Figure 9: Open-loop model mapping external input signals to performance-related outputs.

To build this model, start with the block diagram of the closed-loop system, remove the controller block K, and use `connect` to compute the desired model. As before, the connectivity is specified by labeling the inputs and outputs of each block.

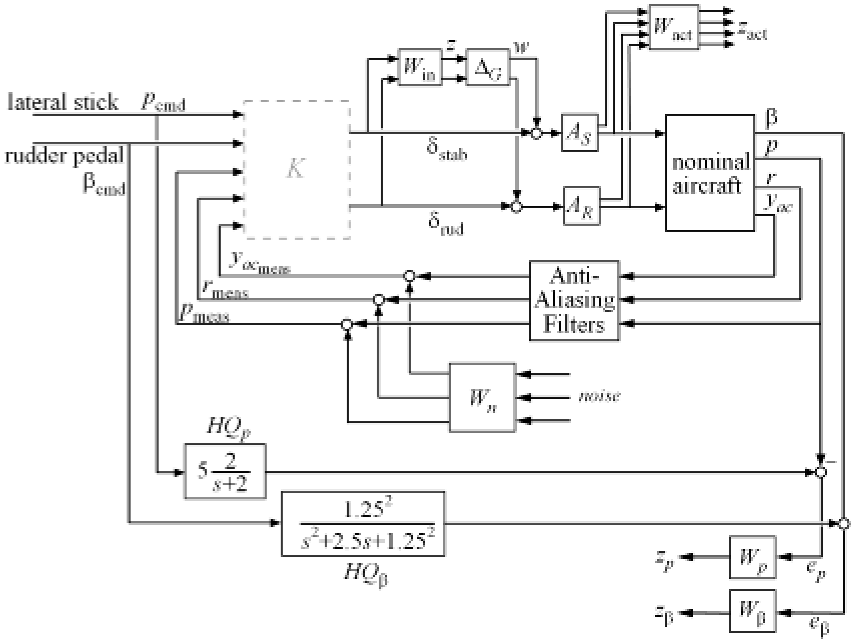


Figure 10: Block diagram for building open-loop model.

```
% Label block I/Os
AAFilters.u = {'p','r','yac'};     AAFilters.y = 'AAFilt';
W_n.u = 'noise';                   W_n.y = 'Wn';
HQ_p.u = 'p_cmd';                  HQ_p.y = 'HQ_p';
HQ_beta.u = 'beta_cmd';            HQ_beta.y = 'HQ_beta';
W_p.u = 'e_p';                     W_p.y = 'z_p';
```

```
W_beta.u = 'e_beta';                W_beta.y = 'z_beta';
W_act.u = [A_S.y ; A_R.y];          W_act.y = 'z_act';

% Specify summing junctions
Sum1 = sumblk('%meas = AAFilt + Wn',{'p_meas','r_meas','yac_meas'});
Sum2 = sumblk('e_p = HQ_p - p');
Sum3 = sumblk('e_beta = HQ_beta - beta');

% Connect everything
OLIC = connect(Plant_unc,AAFilters,W_n,HQ_p,HQ_beta,...
   W_p,W_beta,W_act,Sum1,Sum2,Sum3,...
   {'noise','p_cmd','beta_cmd','delta_stab','delta_rud'},...
   {'z_p','z_beta','z_act','p_cmd','beta_cmd','p_meas','r_meas','yac_meas'});
```

This produces the uncertain state-space model

```
OLIC
```

```
OLIC =

  Uncertain continuous-time state-space model with 11 outputs, 7 inputs, 26 states.
  The model uncertainty consists of the following blocks:
    Delta_G: Uncertain 2x2 LTI, peak gain = 1, 1 occurrences

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

Recall that by construction of the weighting functions, a controller meets the specs whenever the closed-loop gain is less than 1 at all frequencies and for all I/O directions. First design an H-infinity controller that minimizes the closed-loop gain for the nominal aircraft model:

```
nmeas = 5;        % number of measurements
nctrls = 2;        % number of controls
[kinf,~,gamma_inf] = hinfsyn(OLIC.NominalValue,nmeas,nctrls);
gamma_inf
```

```
gamma_inf = 0.9700
```

Here `hinfsyn` computed a controller `kinf` that keeps the closed-loop gain below 1 so the specs can be met for the nominal aircraft model.

Next, perform a mu-synthesis to see if the specs can be met robustly when taking into account the modeling errors (uncertainty `Delta_G`). Use the command `musyn` to perform the synthesis and use `musynOptions` to set the frequency grid used for mu-analysis.

```
fmu = logspace(-2,2,60);
opt = musynOptions('FrequencyGrid',fmu);
[kmu,CLperf] = musyn(OLIC,nmeas,nctrls,opt);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter           K Step        Peak MU       D Fit             D
    1             5.097         3.487        3.488            12
    2              1.31         1.292        1.312            20
    3             1.242         1.242        1.693            12
    4             1.693         1.544        1.545            16
    5             1.223         1.223        1.551            12
    6           1.533        1.464        1.465            20
    7           1.287        1.286        1.303            12

Best achieved robust performance: 1.22
```

```
CLperf
```

```
CLperf = 1.2225
```

Here the best controller kmu cannot keep the closed-loop gain below 1 for the specified model uncertainty, indicating that the specs can be nearly but not fully met for the family of aircraft models under consideration.

Frequency-Domain Comparison of Controllers

Compare the performance and robustness of the H-infinity controller kinf and mu controller kmu. Recall that the performance specs are achieved when the closed loop gain is less than 1 for every frequency. Use the lft function to close the loop around each controller:

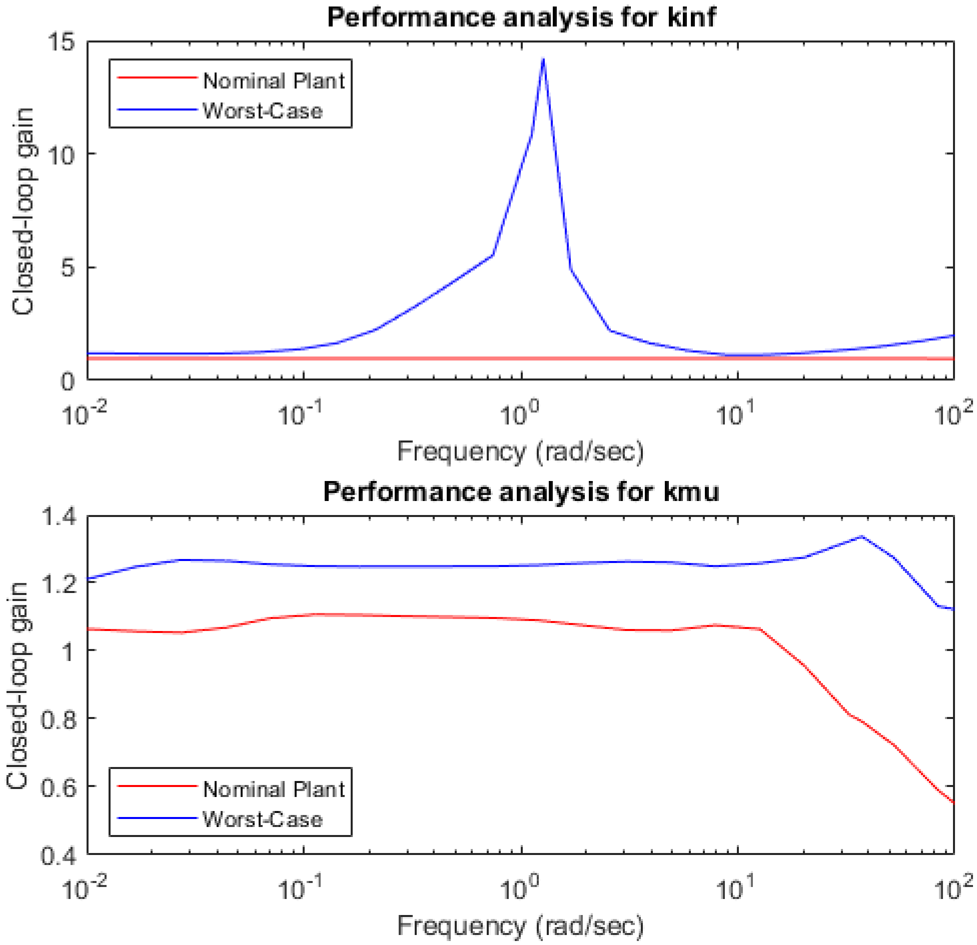
```
clinf = lft(OLIC,kinf);
clmu = lft(OLIC,kmu);
```

What is the worst-case performance (in terms of closed-loop gain) of each controller for modeling errors bounded by W_in? The wcgain command helps you answer this difficult question directly without need for extensive gridding and simulation.

```
% Compute worst-case gain as a function of frequency
opt = wcOptions('VaryFrequency','on');

% Compute worst-case gain (as a function of frequency) for kinf
[mginf,wcuinf,infoinf] = wcgain(clinf,opt);

% Compute worst-case gain for kmu
[mgmu,wcumu,infomu] = wcgain(clmu,opt);
```

You can now compare the nominal and worst-case performance for each controller:

```
clf
subplot(211)
f = infoinf.Frequency;
gnom = sigma(clinf.NominalValue,f);
semilogx(f,gnom(1,:),'r',f,infoinf.Bounds(:,2),'b');
title('Performance analysis for kinf')
xlabel('Frequency (rad/sec)')
ylabel('Closed-loop gain');
xlim([1e-2 1e2])
legend('Nominal Plant','Worst-Case','Location','NorthWest');

subplot(212)
f = infomu.Frequency;
gnom = sigma(clmu.NominalValue,f);
semilogx(f,gnom(1,:),'r',f,infomu.Bounds(:,2),'b');
title('Performance analysis for kmu')
xlabel('Frequency (rad/sec)')
ylabel('Closed-loop gain');
xlim([1e-2 1e2])
legend('Nominal Plant','Worst-Case','Location','SouthWest');
```



The first plot shows that while the H-infinity controller `kinf` meets the performance specs for the nominal plant model, its performance can sharply deteriorate (peak gain near 15) for some perturbed model within our modeling error bounds.

In contrast, the mu controller `kmu` has slightly worse performance for the nominal plant when compared to `kinf`, but it maintains this performance consistently for all perturbed models (worst-case gain near 1.25). The mu controller is therefore more **robust** to modeling errors.

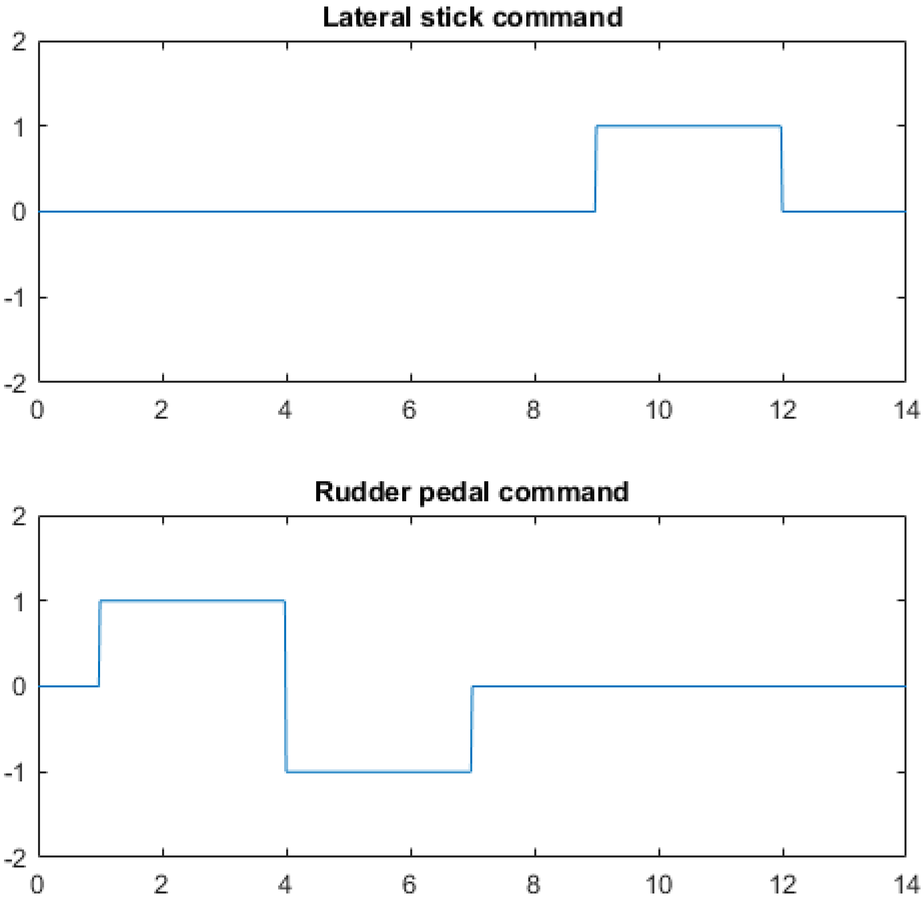
Time-Domain Validation of the Robust Controller

To further test the robustness of the mu controller `kmu` in the time domain, you can compare the time responses of the nominal and worst-case closed-loop models with the ideal "Handling Quality" response. To do this, first construct the "true" closed-loop model `CLSIM` where all weighting functions and HQ reference models have been removed:

```
kmu.u = {'p_cmd','beta_cmd','p_meas','r_meas','yac_meas'};
kmu.y = {'delta_stab','delta_rud'};

AAFilters.y = {'p_meas','r_meas','yac_meas'};

CLSIM = connect(Plant_unc(5:end,:),AAFilters,kmu,{'p_cmd','beta_cmd'},{'p','beta'});
```

Next, create the test signals u_stick and u_pedal shown below

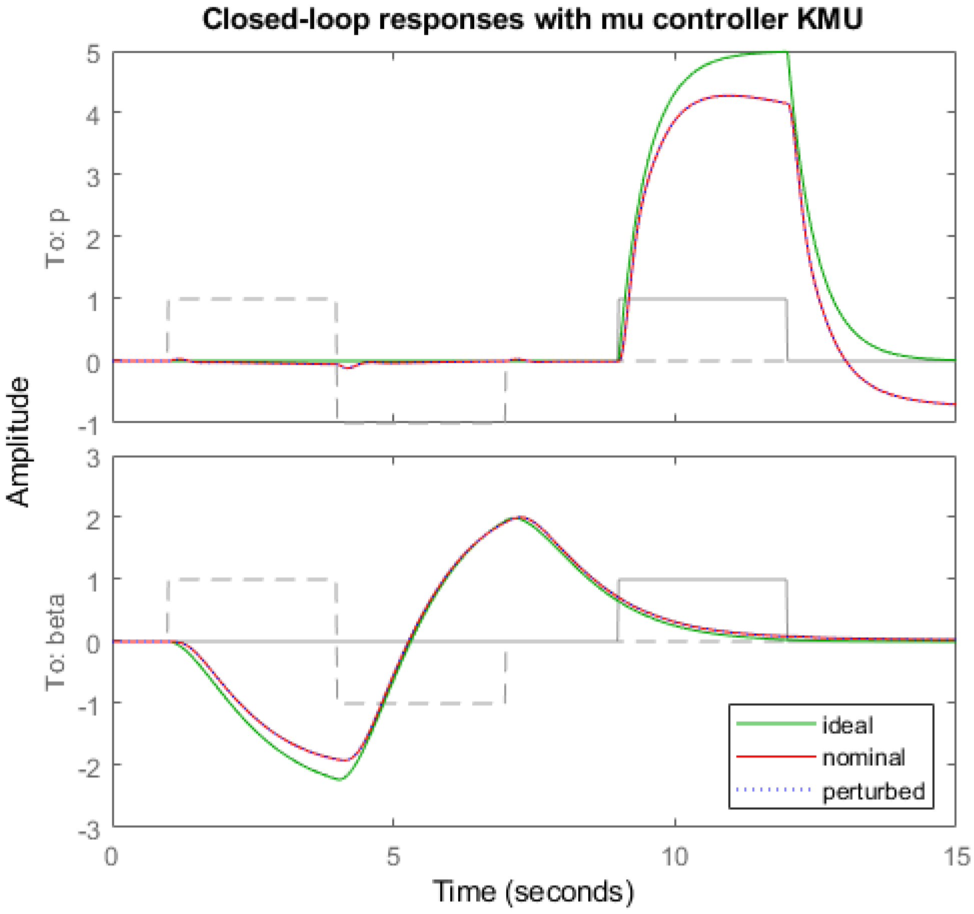
```
time = 0:0.02:15;
u_stick = (time>=9 & time<12);
u_pedal = (time>=1 & time<4) - (time>=4 & time<7);

clf
subplot(211), plot(time,u_stick), axis([0 14 -2 2]), title('Lateral stick command')
subplot(212), plot(time,u_pedal), axis([0 14 -2 2]), title('Rudder pedal command')
```



You can now compute and plot the ideal, nominal, and worst-case responses to the test commands u_stick and u_pedal.

```
% Ideal behavior
IdealResp = append(HQ_p,HQ_beta);
IdealResp.y = {'p','beta'};

% Worst-case response
WCResp = usubs(CLSIM,wcumu);

% Compare responses
clf
```

```
lsim(IdealResp,'g',CLSIM.NominalValue,'r',WCResp,'b:',[u_stick ; u_pedal],time)
legend('ideal','nominal','perturbed','Location','SouthEast');
title('Closed-loop responses with mu controller KMU')
```



The closed-loop response is nearly identical for the nominal and worst-case closed-loop systems. Note that the roll-rate response of the aircraft tracks the roll-rate command well initially and then departs from this command. This is due to a right-half plane zero in the aircraft model at 0.024 rad/sec.

See Also

`hinfsyn` | `musyn`

More About

- “Robust Controller Design Using Mu Synthesis” on page 3-2
- “H-Infinity Performance”
- “Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis” on page 3-77

Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis

This example shows how to perform mixed-mu synthesis with the `musyn` command in the Robust Control Toolbox™. Here `musyn` is used to design a robust controller for a two mass-spring-damper system with uncertainty in the spring stiffness connecting the two masses. This example is taken from the paper "*Robust mixed-mu synthesis performance for mass-spring system with stiffness uncertainty,*" *D. Barros, S. Fekri and M. Athans, 2005 Mediterranean Control Conference*.

Performance Specifications

Consider the mass-spring-damper system in Figure 1. Spring `k2` and damper `b2` are attached to the wall and mass `m2`. Mass `m2` is also attached to mass `m1` through spring `k1` and damper `b1`. Mass 2 is affected by the disturbance force `f2`. The system is controlled via force `f1` acting on mass `m1`.

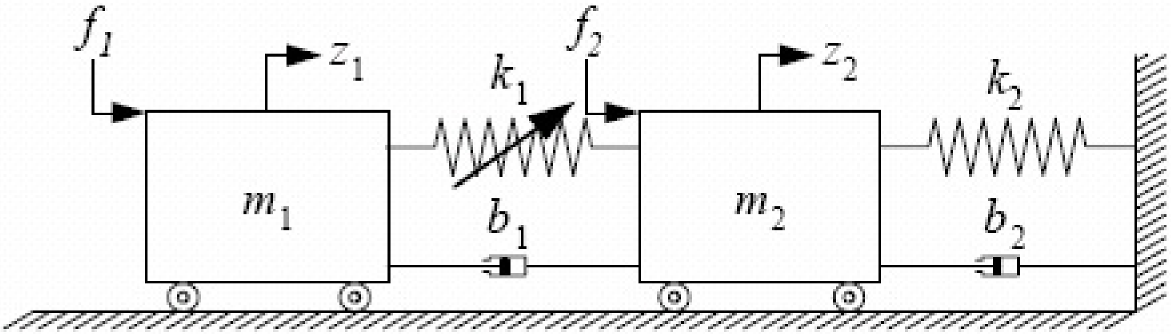


Fig. 1. Mass-spring system.

Our design goal is to use the control force `f1` to attenuate the effect of the disturbance `f2` on the position of mass `m2`. The force `f1` does not directly act on mass `m2`, rather it acts through the spring stiffness `k1`. Hence any uncertainty in the spring stiffness `k1` will make the control problem more difficult. The control problem is formulated as:

- The controller measures the noisy displacement of mass `m2` and applies the control force `f1`. The sensor noise, `Wn`, is modeled as a constant 0.001.
- The actuator command is penalized by a factor 0.1 at low frequency and a factor 10 at high frequency with a crossover frequency of 100 rad/s (filter `Wu`).
- The unit magnitude, first-order coloring filter, `Wdist`, on the disturbance has a pole at 0.25 rad/s.
- The performance objective is to attenuate the disturbance on mass `m2` by a factor of 80 below 0.1 rad/s.

The nominal values of the system parameters are `m1=1`, `m2=2`, `k2=1`, `b1=0.05`, `b2=0.05`, and `k1=2`.

```
Wn = tf(0.001);
Wu = 10*tf([1 10],[1 1000]);
Wdist = tf(0.25,[1 0.25],'inputname','dist','outputname','f2');
Wp = 80*tf(0.1,[1 0.1]);
m1 = 1;
m2 = 2;
k2 = 1;
```

```
b1 = 0.05;
b2 = 0.05;
```

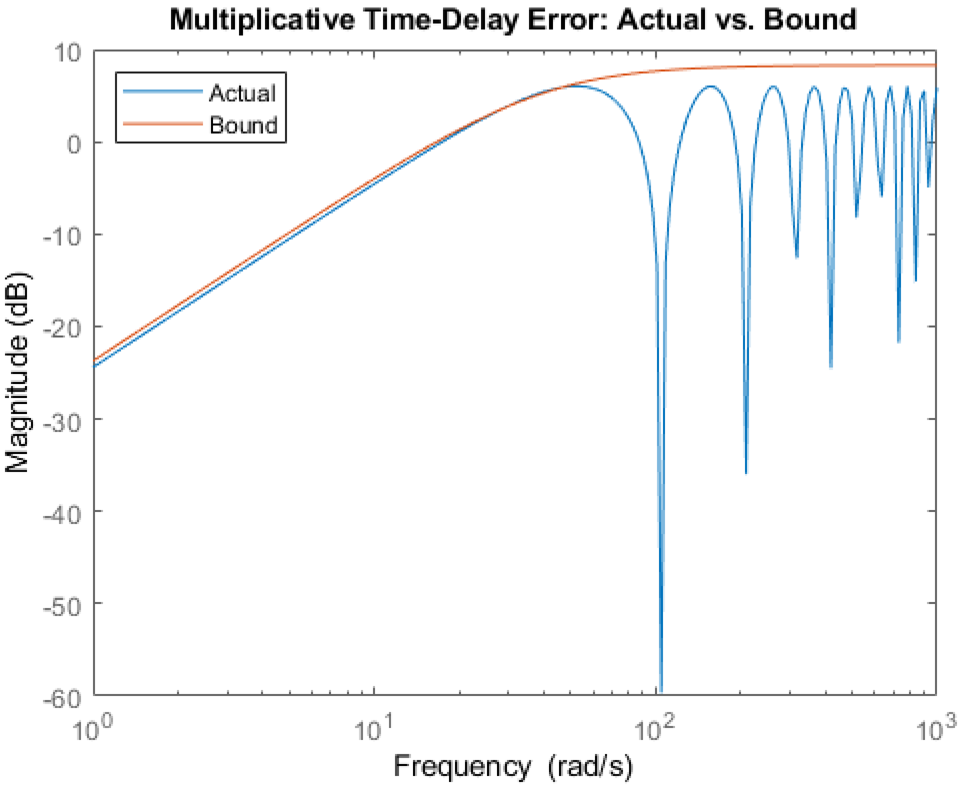
Uncertainty Modeling

The value of spring stiffness `k1` is uncertain. It has a nominal value of 2 and its value can vary between 1.2 and 2.8.

```
k1 = ureal('k1',2,'Range',[1.2 2.8]);
```

There is also a time delay `tau` between the commanded actuator force `f1` and its application to mass `m1`. The maximum delay is 0.06 seconds. Neglecting this time delay introduces a multiplicative error of `exp(-s*tau)-1`. This error can be treated as unmodeled dynamics bounded in magnitude by the high-pass filter `Wunmod = 2.6*s/(s + 40)`:

```
tau = ss(1,'InputDelay',0.06);
Wunmod = 2.6*tf([1 0],[1 40]);
bodemag(tau-1,Wunmod,logspace(0,3,200));
title('Multiplicative Time-Delay Error: Actual vs. Bound')
legend('Actual','Bound','Location','NorthWest')
```



Construct an uncertain state-space model of the plant with the control force `f1` and disturbance `f2` as inputs.

```
a1c = [0 0 -1/m1  1/m2]'*k1;
a2c = [0 0  1/m1 -1/m2]'*k1 + [0 0 0 -k2/m2]';
a3c = [1 0 -b1/m1 b1/m2]';
```

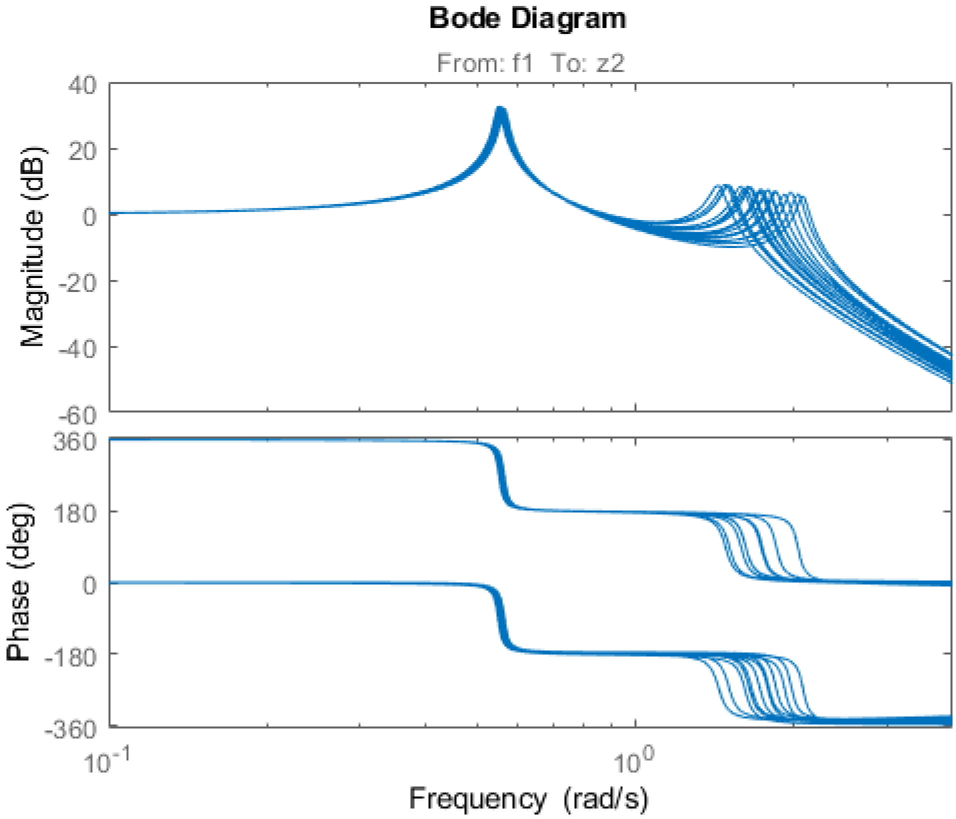
```
a4c = [0 1 b1/m1 -(b1+b2)/m2]';
A  = [a1c a2c a3c a4c];
plant = ss(A,[0 0;0 0;1/m1 0;0 1/m2],[0 1 0 0],[0 0]);
plant.StateName = {'z1';'z2';'z1dot';'z2dot'};
plant.OutputName = {'z2'};
```

Add the unmodeled delay dynamics at the first plant input.

```
Delta = ultidyn('Delta',[1 1]);
plant = plant * append(1+Delta*Wunmod,1);
plant.InputName = {'f1','f2'};
```

Plot the Bode response from f1 to z2 for 20 sample values of the uncertainty. The uncertainty on the value of k1 causes fluctuations in the natural frequencies of the plant modes.

```
bode(plant(1,1),{0.1,4})
```



Control Design

We use the following structure for controller synthesis:

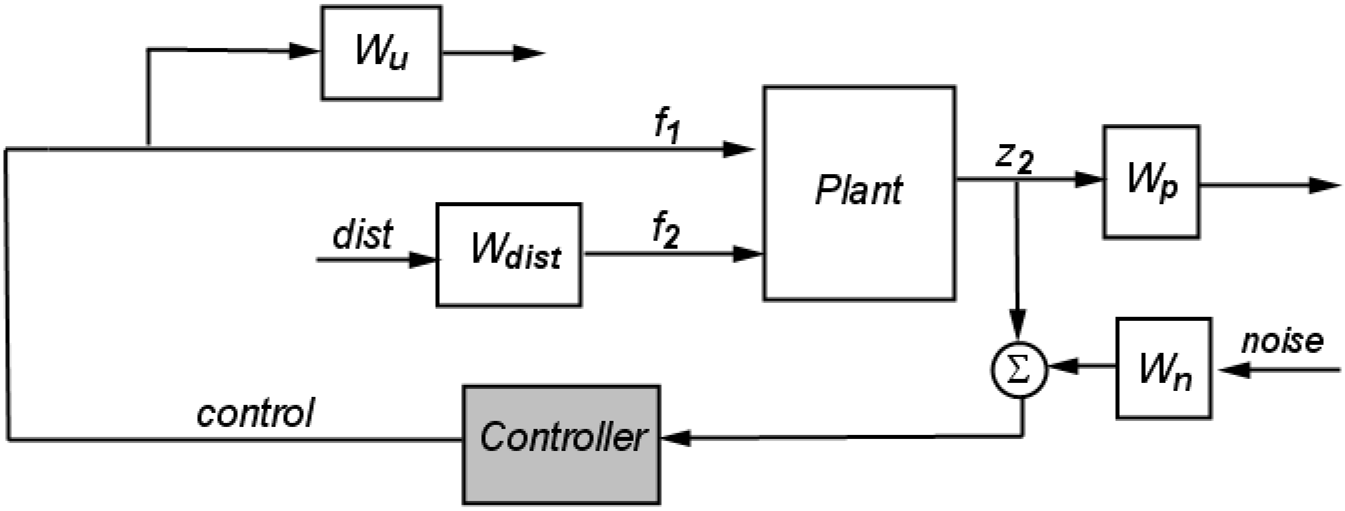


Figure 2

Use `connect` to construct the corresponding open-loop interconnection `IC`. Note that `IC` is an uncertain model with uncertain variables `k1` and `Delta`.

```
Wu.u = 'f1';  Wu.y = 'Wu';
Wp.u = 'z2';  Wp.y = 'Wp';
Wn.u = 'noise';  Wn.y = 'Wn';
S = sumblk('z2n = z2 + Wn');
IC = connect(plant,Wdist,Wu,Wp,Wn,S,{'dist','noise','f1'},{'Wp','Wu','z2n'})
```

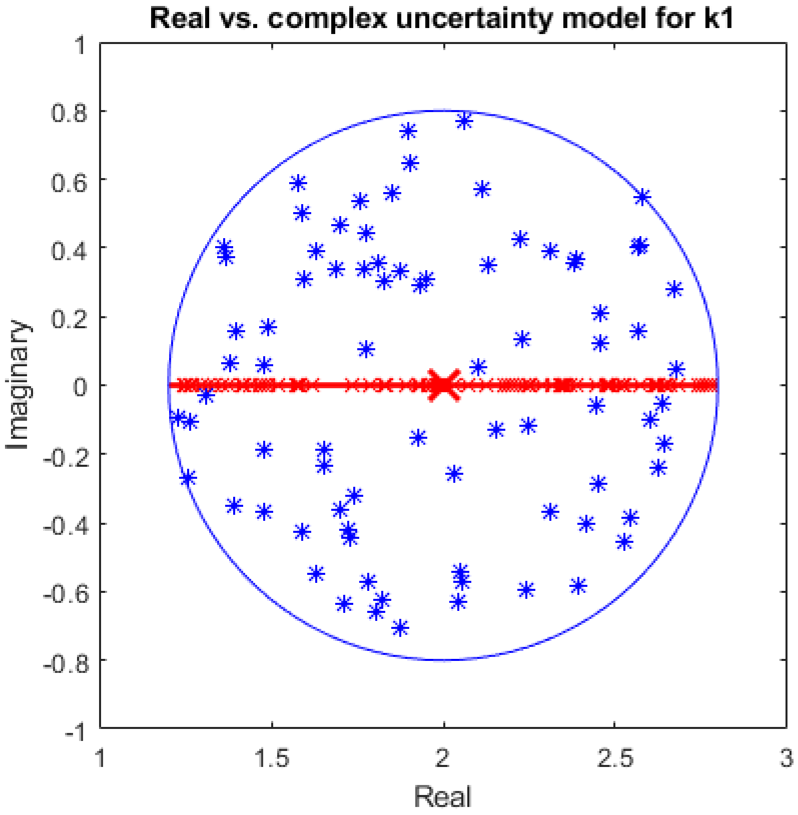
```
IC =

  Uncertain continuous-time state-space model with 3 outputs, 3 inputs, 8 states.
  The model uncertainty consists of the following blocks:
    Delta: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    k1: Uncertain real, nominal = 2, range = [1.2,2.8], 1 occurrences

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

Complex mu-Synthesis

You can use the command `musyn` to synthesize a robust controller for the open-loop interconnection `IC`. By default, `musyn` treats all uncertain real parameters, in this example `k1`, as complex uncertainty. Recall that `k1` is a real parameter with a nominal value of 2 and a range between 1.2 and 2.8. In complex mu-synthesis, it is replaced by a complex uncertain parameter varying in a disk centered at 2 and with radius 0.8. The plot below compares the range of k1 values when k1 is treated as real (red x) vs. complex (blue *).

```
k1c = ucomplex('k1c',2,'Radius',0.8);  % complex approximation

% Plot 80 samples of the real and complex parameters
k1samp = usample(k1,80);
k1csamp = usample(k1c,80);
plot(k1samp(:),0*k1samp(:),'rx',real(k1csamp(:)),imag(k1csamp(:)),'b*')
```

```
hold on

% Draw value ranges for real and complex k1
plot(k1.Nominal,0,'rx',[1.2 2.8],[0 0],'r-','MarkerSize',14,'LineWidth',2)
the=0:0.02*pi:2*pi;
z=sin(the)+sqrt(-1)*cos(the);
plot(real(0.8*z+2),imag(0.8*z),'b')
hold off

% Plot formatting
axis([1 3 -1 1]), axis square
ylabel('Imaginary'), xlabel('Real')
title('Real vs. complex uncertainty model for k1')
```



Synthesize a robust controller Kc using complex mu-synthesis (treating k1 as a complex parameter).

```
[Kc,mu_c,infoc] = musyn(IC,1,1);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step       Peak MU       D Fit             D
    1           2.954        2.455        2.483            16
    2           1.146        1.144        1.154            18
    3           1.086        1.086         1.09            18
    4           1.084        1.083        1.085            18
    5           1.082        1.081        1.082            18

Best achieved robust performance: 1.08
```

Note that mu_c exceeds 1 so the controller Kc fails to robustly achieve the desired performance level.

Mixed-Mu Synthesis

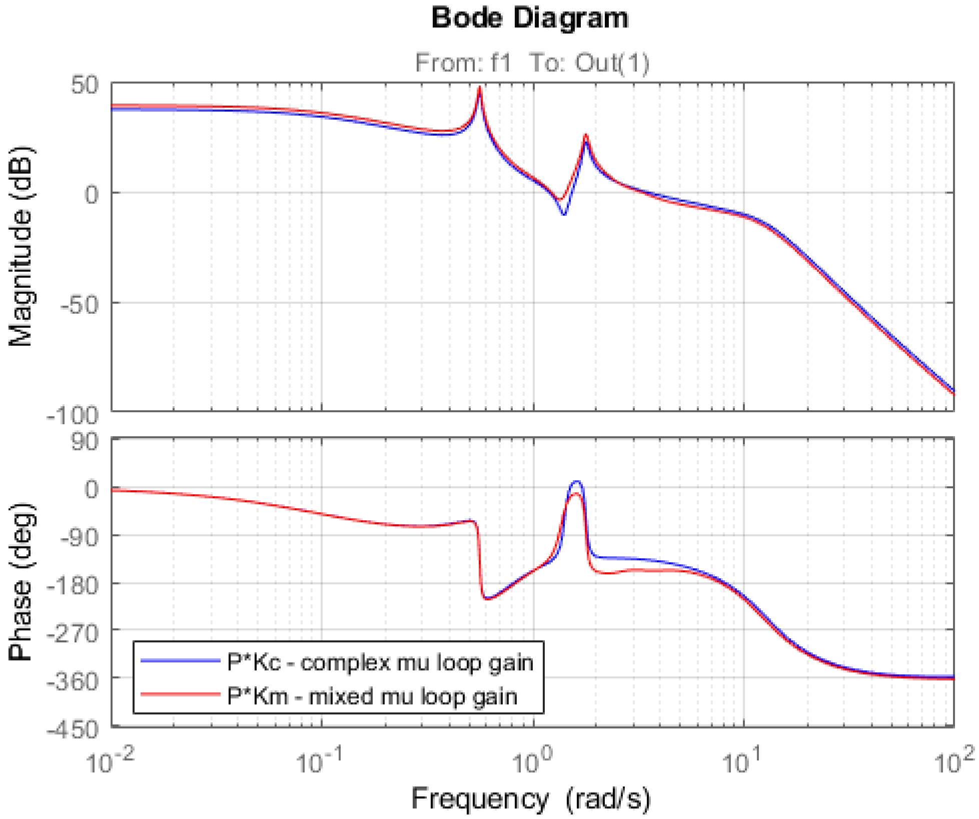
Mixed-mu synthesis accounts for uncertain real parameters directly in the synthesis process. Enable mixed-mu synthesis by setting the MixedMU option to 'on'.

```
opt = musynOptions('MixedMU','on');
[Km,mu_m] = musyn(IC,1,1,opt);
```

```
DG-K ITERATION SUMMARY:
-------------------------------------------------------------------
                       Robust performance                 Fit order
-------------------------------------------------------------------
  Iter          K Step        Peak MU        DG Fit           D       G
    1            2.954         2.081         2.371           16       8
    2            1.674         1.433         1.736           14       8
    3            0.966         1.019          1.23           20       8
    4           0.9125        0.9536         1.002           20       8
    5           0.8993        0.9291        0.9872           18       8
    6           0.8978        0.9128        0.9368           20       8
    7           0.8917        0.9064        0.9462           20       8
    8           0.8881        0.8994        0.9514           18       8
    9           0.8874         0.893        0.9592           18       8
   10           0.8825         0.892        0.9529           20       8

Best achieved robust performance: 0.892
```

Mixed-mu synthesis is able to find a controller that achieves the desired performance and robustness objectives. A comparison of the open-loop responses shows that the mixed-mu controller Km gives less phase margin near 3 rad/s because it only needs to guard against real variations of k1.

```
clf
% Note: Negative sign because interconnection in Fig 2 uses positive feedback
bode(-Kc*plant.NominalValue(1,1),'b',-Km*plant.NominalValue(1,1),'r',{1e-2,1e2})
grid
legend('P*Kc - complex mu loop gain','P*Km - mixed mu loop gain','location','SouthWest')
```



Worst-Case Analysis

A comparison of the two controllers indicates that taking advantage of the "realness" of `k1` results in a better performing, more robust controller.

To assess the worst-case closed-loop performance of `Kc` and `Km`, form the closed-loop interconnection of Figure 2 and use the command `wcgain` to determine how large the disturbance-to-error norm can get for the specified plant uncertainty.

```
clpKc = lft(IC,Kc);
clpKm = lft(IC,Km);
[maxgainKc,badpertKc] = wcgain(clpKc);
maxgainKc
```

```
maxgainKc = 

  struct with fields:

           LowerBound: 2.0852
           UpperBound: 2.0894
    CriticalFrequency: 1.4295
```

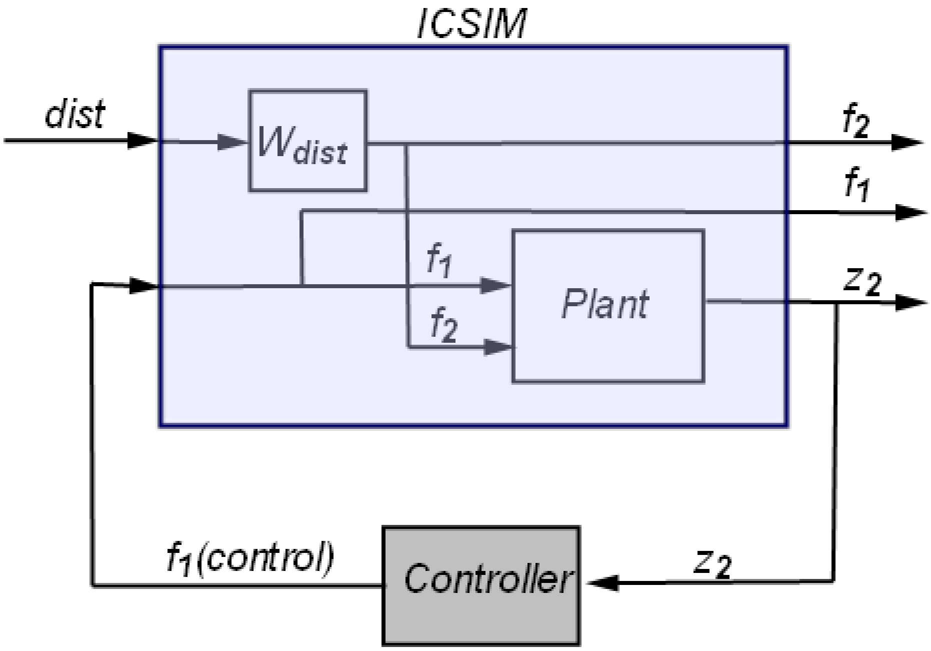
```
[maxgainKm,badpertKm] = wcgain(clpKm);
maxgainKm
```

```
maxgainKm = 

  struct with fields:

           LowerBound: 0.8810
           UpperBound: 0.8827
    CriticalFrequency: 0.1439
```

The mixed-mu controller Km has a worst-case gain of 0.88 while the complex-mu controller Kc has a worst-case gain of 2.2, or 2.5 times larger.

Disturbance Rejection Simulations

To compare the disturbance rejection performance of Kc and Km, first build closed-loop models of the transfer from input disturbance dist to f2, f1, and z2 (position of the mass m2)

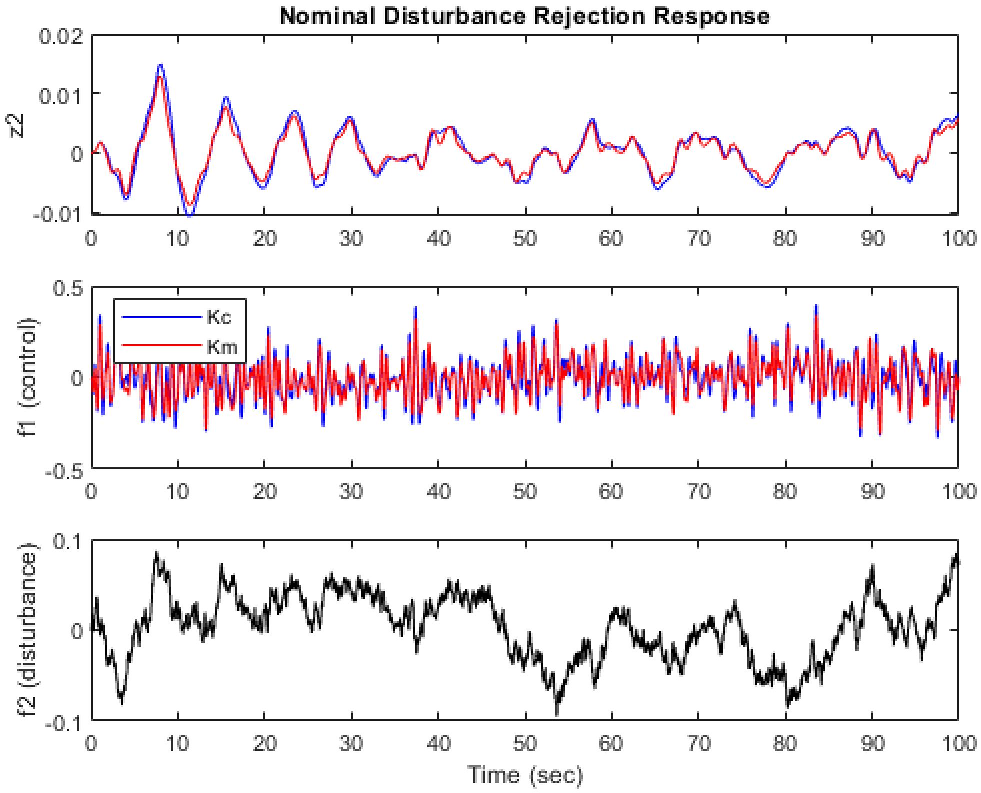


```
Km.u = 'z2';  Km.y = 'f1';
clsimKm = connect(plant,Wdist,Km,'dist',{'f2','f1','z2'});
Kc.u = 'z2';  Kc.y = 'f1';
clsimKc = connect(plant,Wdist,Kc,'dist',{'f2','f1','z2'});
```

Inject white noise into the low-pass filter Wdist to simulate the input disturbance f2. The nominal closed-loop performance of the two designs is nearly identical.

```
t = 0:.01:100;
dist = randn(size(t));
yKc = lsim(clsimKc.Nominal,dist,t);
yKm = lsim(clsimKm.Nominal,dist,t);

% Plot
subplot(311)
plot(t,yKc(:,3),'b',t,yKm(:,3),'r')
```

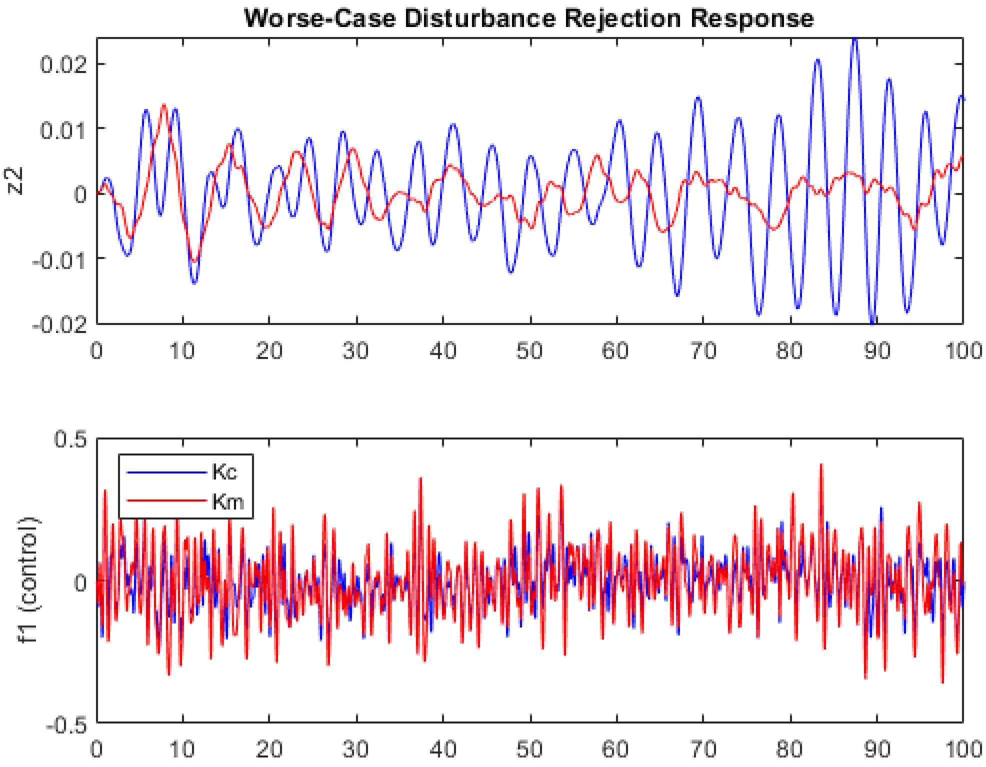
```
title('Nominal Disturbance Rejection Response')
ylabel('z2')

subplot(312)
plot(t,yKc(:,2),'b',t,yKm(:,2),'r')
ylabel('f1 (control)')
legend('Kc','Km','Location','NorthWest')

subplot(313)
plot(t,yKc(:,1),'k')
ylabel('f2 (disturbance)')
xlabel('Time (sec)')
```



Next, compare the worst-case scenarios for Kc and Km by setting the plant uncertainty to the worst-case values computed with wcgain.

```
clsimKc_wc = usubs(clsimKc,badpertKc);
clsimKm_wc = usubs(clsimKm,badpertKm);
yKc_wc = lsim(clsimKc_wc,dist,t);
yKm_wc = lsim(clsimKm_wc,dist,t);

subplot(211)
plot(t,yKc_wc(:,3),'b',t,yKm_wc(:,3),'r')
title('Worse-Case Disturbance Rejection Response')
ylabel('z2')
subplot(212)
plot(t,yKc_wc(:,2),'b',t,yKm_wc(:,2),'r')
```

```
ylabel('f1 (control)')
legend('Kc','Km','Location','NorthWest')
```



This shows that the mixed-mu controller Km significantly outperforms Kc in the worst-case scenario. By exploiting the fact that k1 is real, the mixed-mu controller is able to deliver better performance at equal robustness.

Controller Simplification

The mixed-mu controller Km has relatively high order compared to the plant. To obtain a simpler controller, use musyn's fixed-order tuning capability. This uses hinfstruct instead of hinfsyn for the synthesis step. You can try different orders to find the simplest controller that maintains robust performance. For example, try tuning a fifth-order controller. Use the "RandomStart" option to run several mu-synthesis cycles, each starting from a different initial value of K.

```
K = tunableSS('K',5,1,1);  % 5th-order tunable state-space model

opt = musynOptions('MixedMU','on','MaxIter',20,'RandomStart',2);
rng(0), [CL,mu_f] = musyn(lft(IC,K),opt);
```

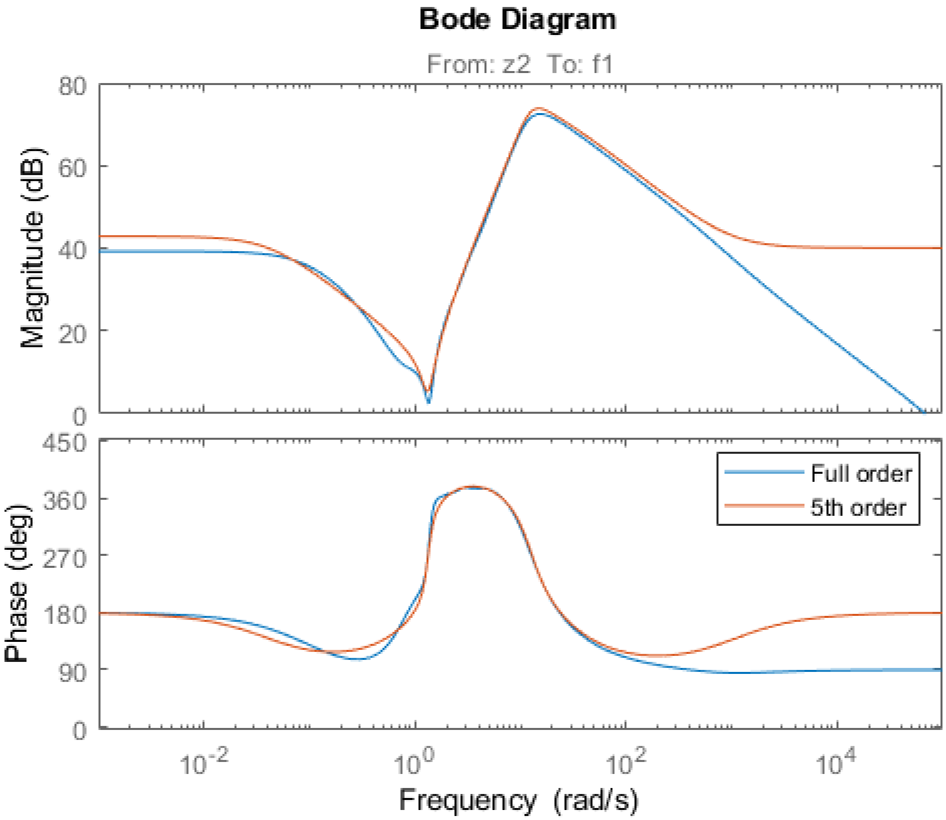
```
=== Synthesis 1 of 3 ============================================


DG-K ITERATION SUMMARY:
-------------------------------------------------------------------
                       Robust performance                  Fit order
```

```
-------------------------------------------------------------------
  Iter         K Step       Peak MU       DG Fit           D      G
    1           43.48        30.74        31.06           12      6
    2           9.524        10.01         11.6           18      6
    3           9.008        8.909            9           16      6
    4           6.209        6.997           19           20      6
    5           6.993        6.897        7.074           12      8
    6           4.184        4.131        4.169           18      8
    7           2.716        2.987        6.471           16      8
    8           2.388        2.336        2.355           18      8
    9            1.47        1.968         3.04           20      8
   10            1.64        1.623        1.699           20      8
   11           1.285        1.313        1.732           20      8
   12           1.327        1.326        1.343           20      8
   13            1.23        1.228        1.445           20      8
   14           1.244        1.242        1.248           20      8
   15           1.206        1.204        1.473           20      8
   16           1.172        1.171        1.186           20      8
   17           1.124        1.122        1.337           18      8
   18           1.137        1.127        1.138           20      8
   19           1.094        1.093         1.17           18      8
   20           1.062        1.061        1.098           18      8

Best achieved robust performance: 1.06


=== Synthesis 2 of 3 ============================================


DG-K ITERATION SUMMARY:
-------------------------------------------------------------------
                       Robust performance                 Fit order
-------------------------------------------------------------------
  Iter         K Step       Peak MU       DG Fit           D      G
    1           3.166        2.165        2.417           18      8
    2           1.618        1.609         1.62           20      8
    3           1.145        1.144        1.209           16      8
    4           1.012        1.025         1.14           18      8
    5           1.029        1.029        1.029           18      8
    6           1.016        1.016        1.045           18      8

Best achieved robust performance: 1.02


=== Synthesis 3 of 3 ============================================


DG-K ITERATION SUMMARY:
-------------------------------------------------------------------
                       Robust performance                 Fit order
-------------------------------------------------------------------
  Iter         K Step       Peak MU       DG Fit           D      G
    1           3.469        3.356        3.357           16      8
    2           2.111          2.1          2.1           18      8
    3           1.706        1.705        2.399           16      8
    4           1.262        1.262        1.262           18      8
    5           1.069        1.072        1.188           16      8
    6           1.036        1.036        1.128           18      8
```

```
     7          1.028       1.026       1.052         18      8
     8          1.024       1.023        1.07         18      8
     9          1.031        1.03       1.041         20      8

Best achieved robust performance: 1.02
```

The best controller nearly delivers the desired robust performance (robust performance `mu_f` is close to 1). Compare the two controllers.

```
clf, bode(Km,getBlockValue(CL,'K'))
legend('Full order','5th order')
```



See Also

`musyn` | `wcgain`

More About

- "Robust Controller Design Using Mu Synthesis" on page 3-2
- "Robustness and Worst-Case Analysis" on page 2-21
- "Control of a Two-Tank System" on page 3-34

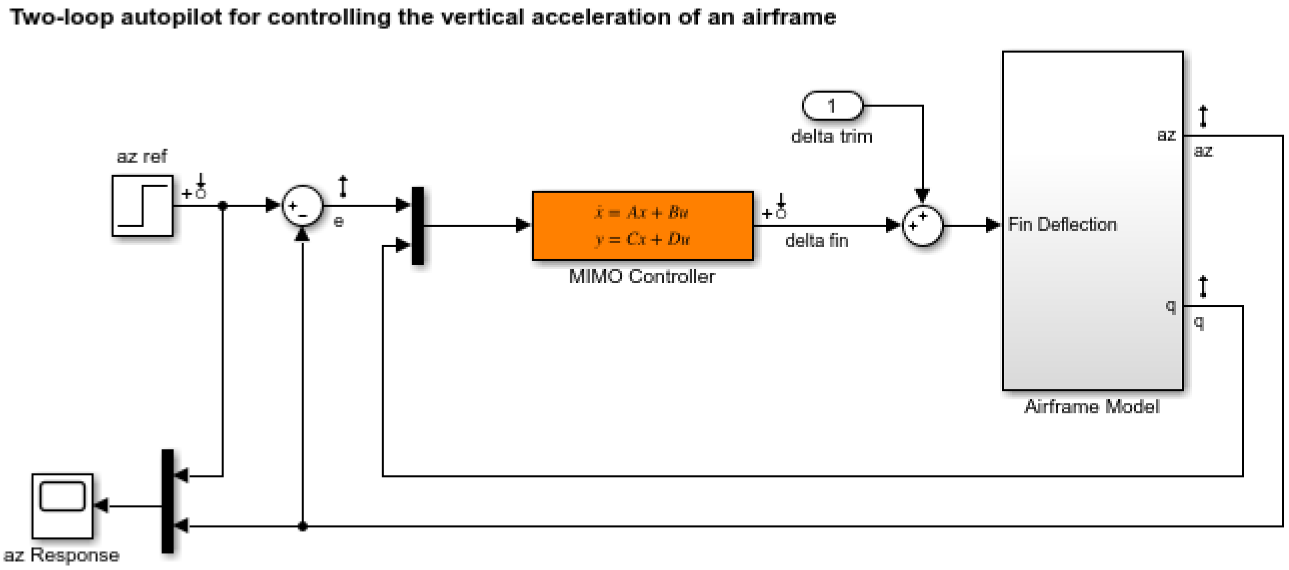
Robust MIMO Controller for Two-Loop Autopilot

This example shows how to design a robust controller for a two-loop autopilot that controls the pitch rate and vertical acceleration of an airframe. The controller is robust against gain and phase variations in the multichannel feedback loop.

Linearized Airframe Model

The airframe dynamics and the autopilot are modeled in Simulink. See “Tuning of a Two-Loop Autopilot” (Control System Toolbox) for more information about this model and for additional design and tuning options.

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


Two-loop autopilot for controlling the vertical acceleration of an airframe

As in the example Tuning of a Two-Loop Autopilot, trim the airframe for $\alpha = 0$ and $V = 984m/s$. The trim condition corresponds to zero normal acceleration and pitching moment (w and q steady). Use `findop` to compute the corresponding operating condition. Then, linearize the airframe model at this trim condition.

```
opspec = operspec('rct_airframe2');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
```

```
% controller states unknown, not steady
opspec.States(5).SteadyState = [0;0];

op = findop('rct_airframe2',opspec);
G = linearize('rct_airframe2','rct_airframe2/Airframe Model',op);
G.InputName = 'delta';
G.OutputName = {'az','q'};
```

```
 Operating point search report:
---------------------------------

 Operating point search report for the Model rct_airframe2.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body /
      x:              0      dx:           984
      x:      -3.05e+03      dx:             0
(2.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body /
      x:              0      dx:      -0.00972
(3.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body /
      x:            984      dx:          22.7
      x:              0      dx:      2.46e-11 (0)
(4.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body /
      x:       -0.00972      dx:     -1.72e-16 (0)
(5.) rct_airframe2/MIMO Controller
      x:       0.000654      dx:        -0.009
      x:       4.13e-19      dx:        0.0303

Inputs:
----------
(1.) rct_airframe2/delta trim
      u:       0.000436    [-Inf Inf]

Outputs: None
----------
```

Nominal Controller Design

Design an H-infinity controller that responds to a step change in vertical acceleration under 1 second. Use a mixed-sensitivity formulation with a lowpass weight wS that penalizes low-frequency tracking error and a highpass weight wT that enforces adequate roll-off. Build an augmented plant with azref,delta as inputs and the filtered wS*e,wT*az,e,q as outputs. (For information about the mixed-sensitivity formulation, see “Mixed-Sensitivity Loop Shaping”.)

```
sb = sumblk('e = azref-az');
wS = makeweight(1e2,5,0.1);
wS.u = 'e';
wS.y = 'we';
wT = makeweight(0.1,50,1e2);
wT.u = 'az';
wT.y = 'waz';

Paug = connect(G,wS,wT,sb,{'azref','delta'},{'we','waz','e','q'});
```

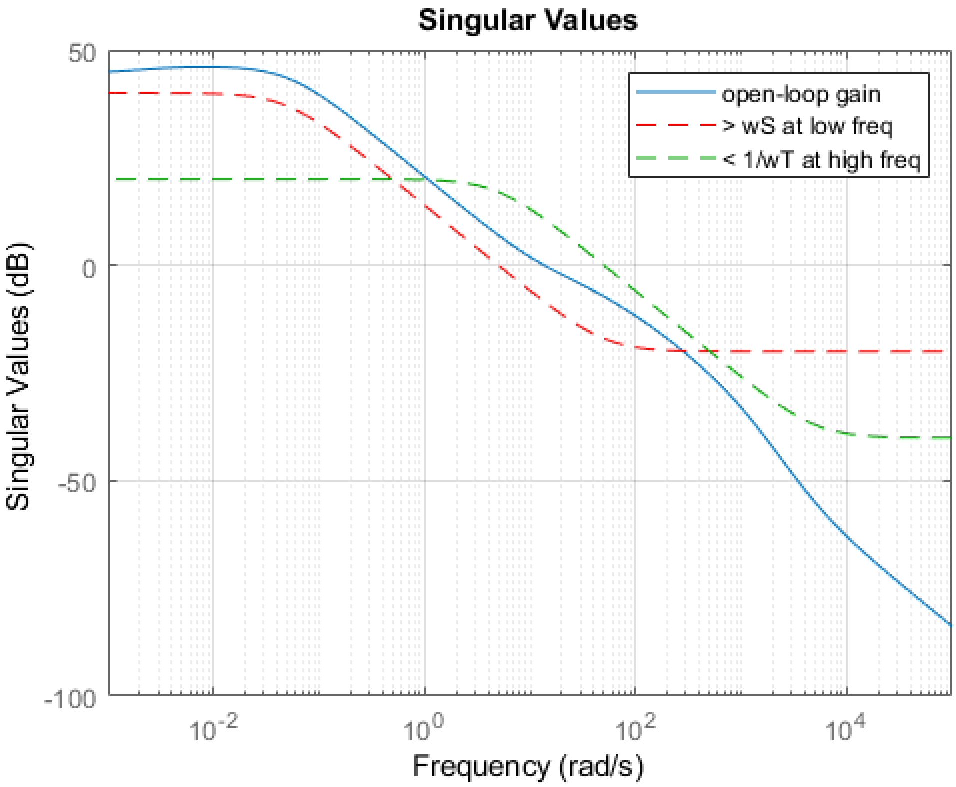
Compute the optimal H-infinity controller using `hinfsyn`. In this configuration there are two measurements `e,q` and one control `delta`.

```
[Knom,~,gam] = hinfsyn(Paug,2,1);
gam
```

```
gam =

    0.5928
```

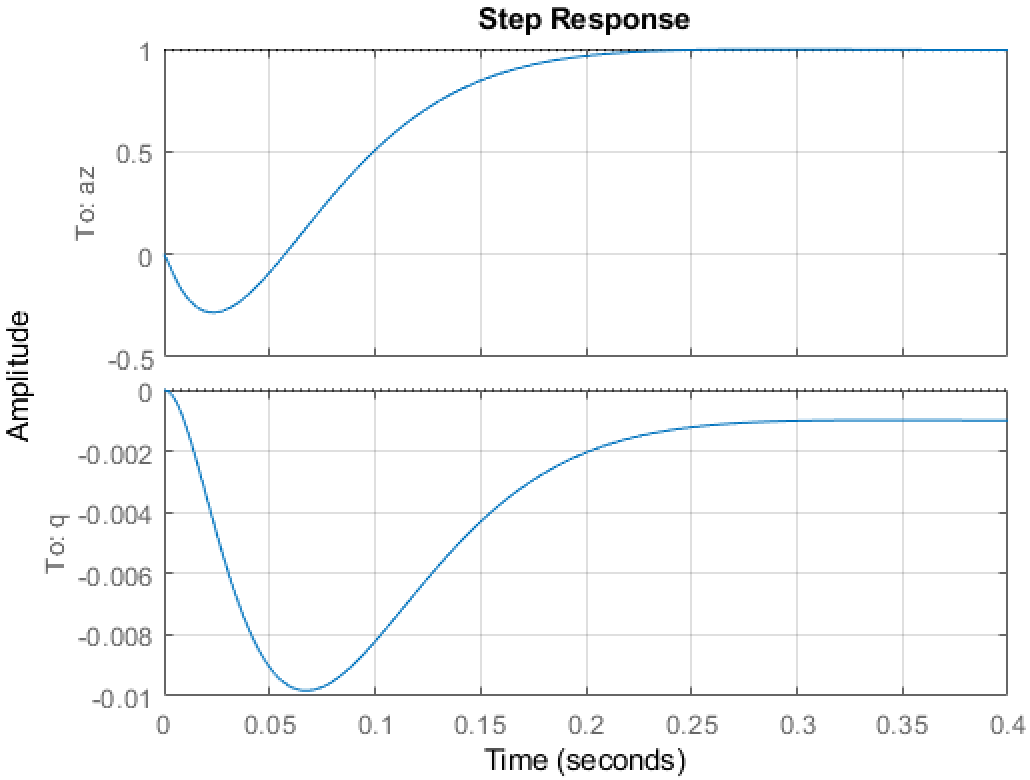
Verify the open-loop gain against `wS,wT`.

```
f = figure();
sigma(Knom*G,wS,'r--',1/wT,'g--'), grid
legend('open-loop gain','> wS at low freq','< 1/wT at high freq')
```



Plot the closed-loop response.

```
CL = feedback(G*Knom,diag([1 -1]));
step(CL(:,1)), grid
```



Robustness Analysis

Compute the disk margins at the plant input and outputs. That the `az` loop uses negative feedback while the `q` loop uses positive feedback, so change the sign of the loop gain of the `q` loop before using `diskmargin`.

```
loopsgn = diag([1 -1]);
```

Examine the margins at the plant input.

```
DM = diskmargin(Knom*loopsgn*G)
```

```
DM = 

  struct with fields:

           GainMargin: [0.3918 2.5524]
          PhaseMargin: [-47.2105 47.2105]
           DiskMargin: 0.8740
           LowerBound: 0.8740
           UpperBound: 0.8740
            Frequency: 28.8842
    WorstPerturbation: [1x1 ss]
```

For the margins at plant outputs, use the multiloop margin to account for simultaneous, independent variations in both output channels.

```
[~,MM] = diskmargin(loopsgn*G*Knom)
```

```
MM = 

  struct with fields:

           GainMargin: [0.4994 2.0025]
          PhaseMargin: [-36.9261 36.9261]
           DiskMargin: 0.6678
           LowerBound: 0.6678
           UpperBound: 0.6691
            Frequency: 23.6855
    WorstPerturbation: [2x2 ss]
```

Finally, compute the margins against simultaneous variations at the plant input and outputs.

```
MMIO = diskmargin(loopsgn*G,Knom)
```

```
MMIO = 

  struct with fields:

           GainMargin: [0.6866 1.4565]
          PhaseMargin: [-21.0565 21.0565]
           DiskMargin: 0.3717
           LowerBound: 0.3717
           UpperBound: 0.3725
            Frequency: 24.8030
    WorstPerturbation: [1x1 struct]
```

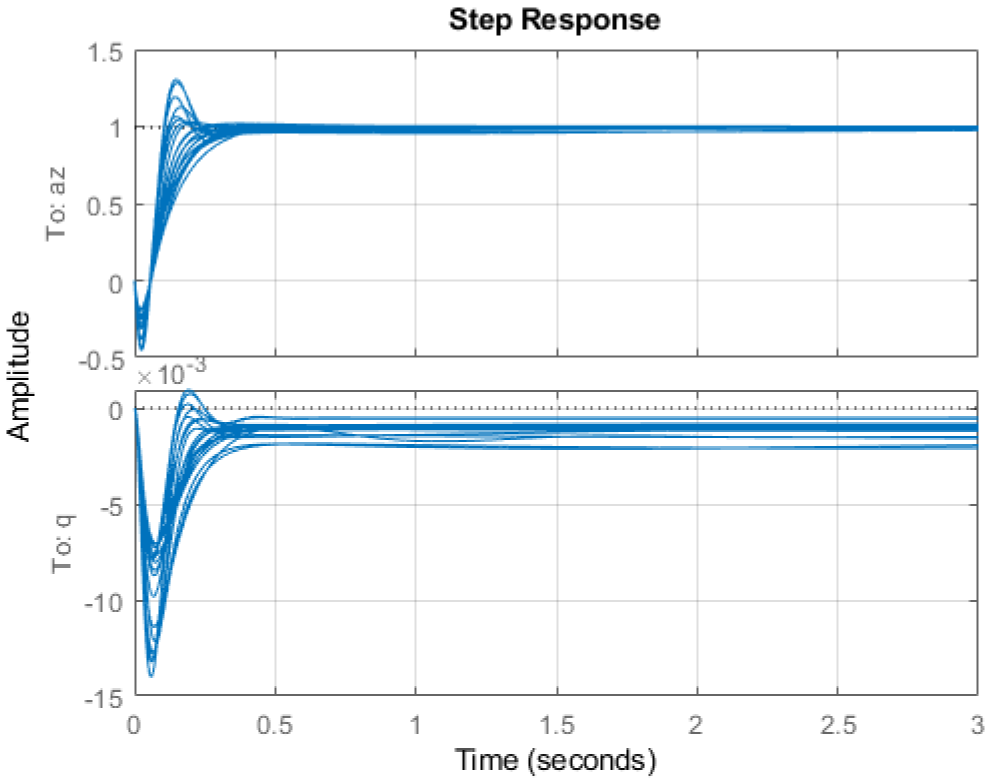
The disk-based gain and phase margins exceed 2 (6dB) and 35 degrees at the plant input and the plant outputs. Moreover, `MMIO` indicates that this feedback loop can withstand gain variations by a factor 1.45 or phase variations of 21 degrees affecting all plant inputs and outputs at once.

Next, investigate the impact of gain and phase uncertainty on performance. Use the `umargin` control design block to model a gain change factor of 1.4 (3dB) and phase change of 20 degrees in each feedback channel. Use `getDGM` to fit an uncertainty disk to these amounts of gain and phase change.

```
GM = 1.4;
PM = 20;
DGM = getDGM(GM,PM,'balanced');
ue = umargin('e',DGM);
uq = umargin('q',DGM);
Gunc = blkdiag(ue,uq)*G;
Gunc.OutputName = {'az','q'};
```

Rebuild the closed-loop model and sample the gain and phase uncertainty to gauge the impact on the step response.

```
CLunc = feedback(Gunc*Knom,loopsgn);
step(CLunc(:,1),3)
grid
```



The plot shows significant variability in performance, with large overshoot and steady-state error for some combinations of gain and phase variations.

Robust Controller Synthesis

Redo the controller synthesis to account for gain and phase variations using `musyn`. This synthesis optimizes performance uniformly for the specified range of gain and phase uncertainty.

```
Punc = connect(Gunc,wS,wT,sb,{'azref','delta'},{'we','waz','e','q'});
[Kr,gam] = musyn(Punc,2,1);
gam
```

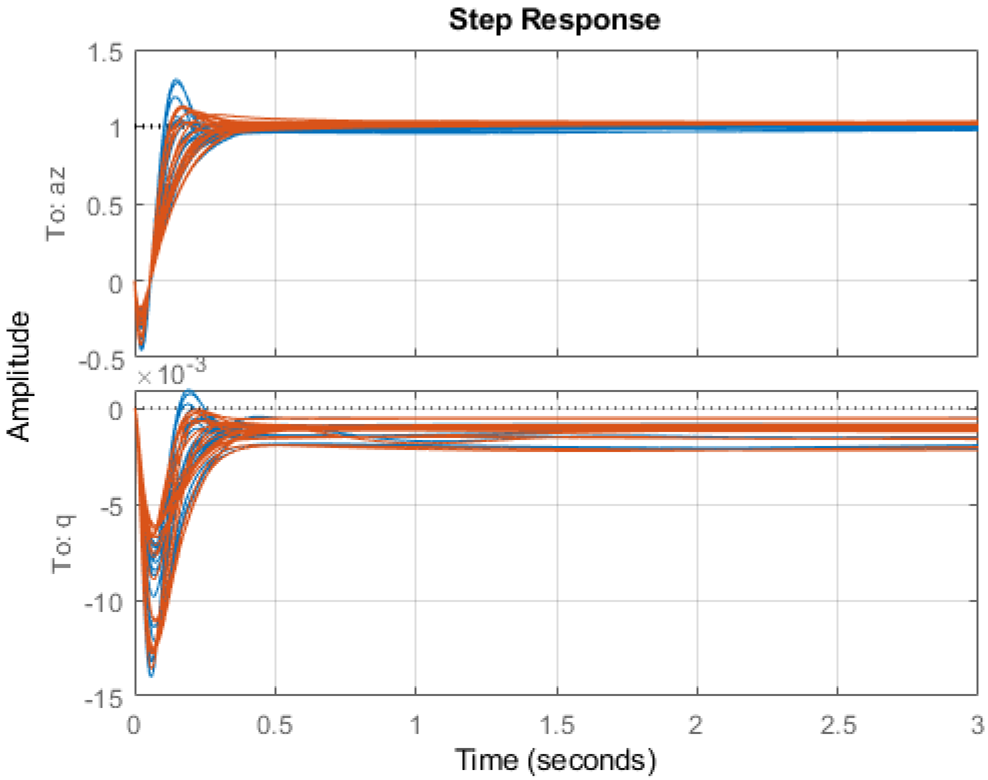
```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step        Peak MU       D Fit             D
    1            51.06         26.33        26.62             4
    2            7.028          5.24        5.303             8
    3            1.681         1.156        1.157             4
    4           0.9705        0.9702       0.9822            10
    5            0.962        0.9619       0.9622            10
    6           0.9625        0.9623       0.9629            10

Best achieved robust performance: 0.962
```

```
gam =

    0.9619
```

Compare the sampled step responses for the nominal and robust controllers. The robust design reduces both overshoot and steady-state errors and gives more consistent performance across the modeled range of gain and phase uncertainty.

```
CLr = feedback(Gunc*Kr,loopsgn);
rng(0) % for reproducibility
step(CLunc(:,1),3)
hold
rng(0)
step(CLr(:,1),3)
grid
```

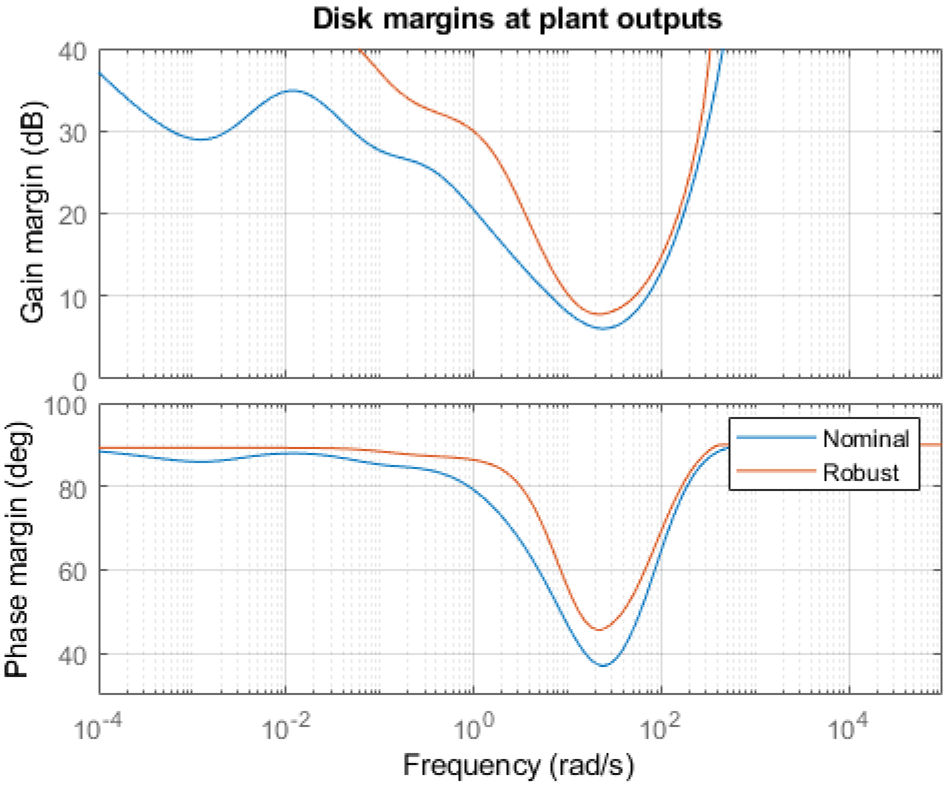
```
Current plot held
```



The robust controller achieves this performance by increasing the (nominal) disk margins at the plant output and, to a lesser extent, the I/O disk margin. For instance, compare the disk-based margins at the plant outputs for the nominal and robust designs. Use `diskmarginplot` to see the variations of the margins with frequency.

```
Lnom = loopsgn*G*Knom;
Lrob = loopsgn*G*Kr;
```

```
clf
diskmarginplot(Lnom,Lrob)
title('Disk margins at plant outputs')
grid
legend('Nominal','Robust')
```



Examine the margins against variations simultaneous variations at the plant inputs and outputs with the robust controller.

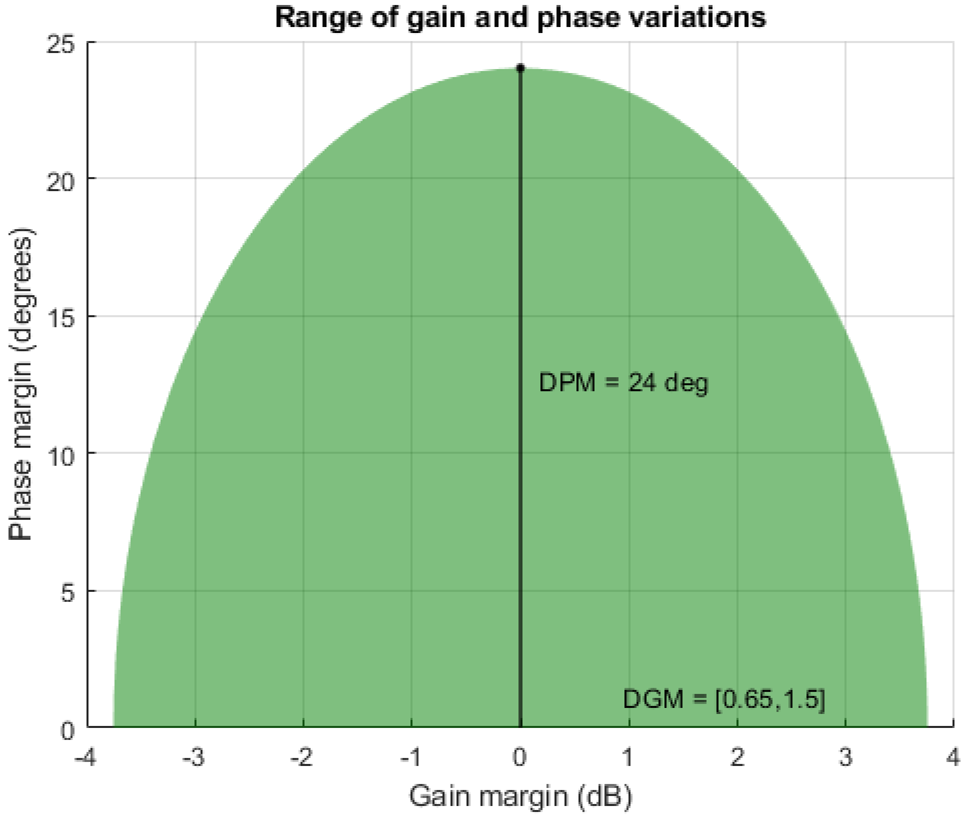
```
MMIO = diskmargin(loopsgn*G,Kr)
```

```
MMIO = 

  struct with fields:

           GainMargin: [0.6492 1.5404]
          PhaseMargin: [-24.0166 24.0166]
           DiskMargin: 0.4254
           LowerBound: 0.4254
           UpperBound: 0.4263
            Frequency: 19.6040
    WorstPerturbation: [1x1 struct]
```

Recall that with the nominal controller, the feedback loop could withstand gain variations of a factor of 1.45 or phase variations of 21 degrees affecting all plant inputs and outputs at once. With the robust controller, those margins increase to about 1.54 and 24 degrees.

View the range of simultaneous gain and phase variations that the robust design can sustain at all plant inputs and outputs.

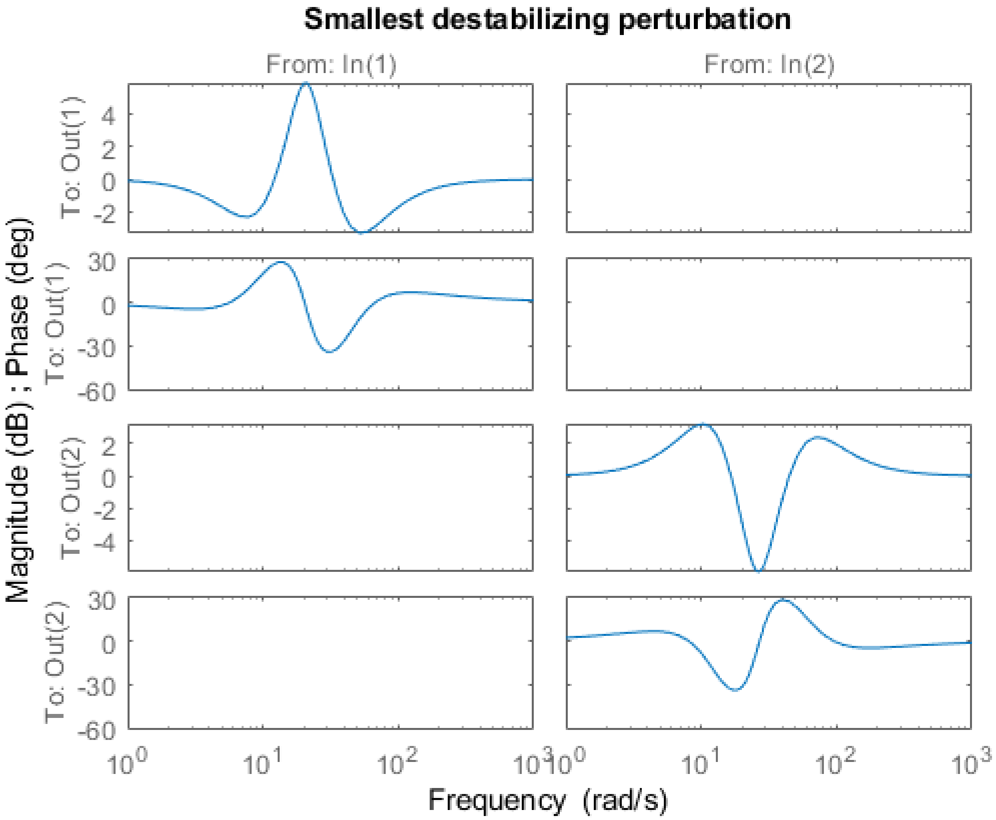
```
diskmarginplot(MMIO.GainMargin)
```



Nonlinear Simulation of Worst-Case Scenario

`diskmargin` returns the smallest change in gain and phase that destabilizes the feedback loop. It can be insightful to inject this perturbation in the nonlinear simulation to further analyze the implications for the real system. For example, compute the destabilizing perturbation at the plant outputs for the nominal controller.

```
[~,MM] = diskmargin(Lnom);
WP = MM.WorstPerturbation;
bode(WP)
title('Smallest destabilizing perturbation')
```



The worst perturbation is a diagonal, dynamic perturbation that multiplies the open-loop response at the plant outputs. With this perturbation, the closed-loop system becomes unstable with an undamped pole at the frequency `w0 = MM.Frequency`.

```
damp(feedback(WP*G*Knom,loopsgn))
```

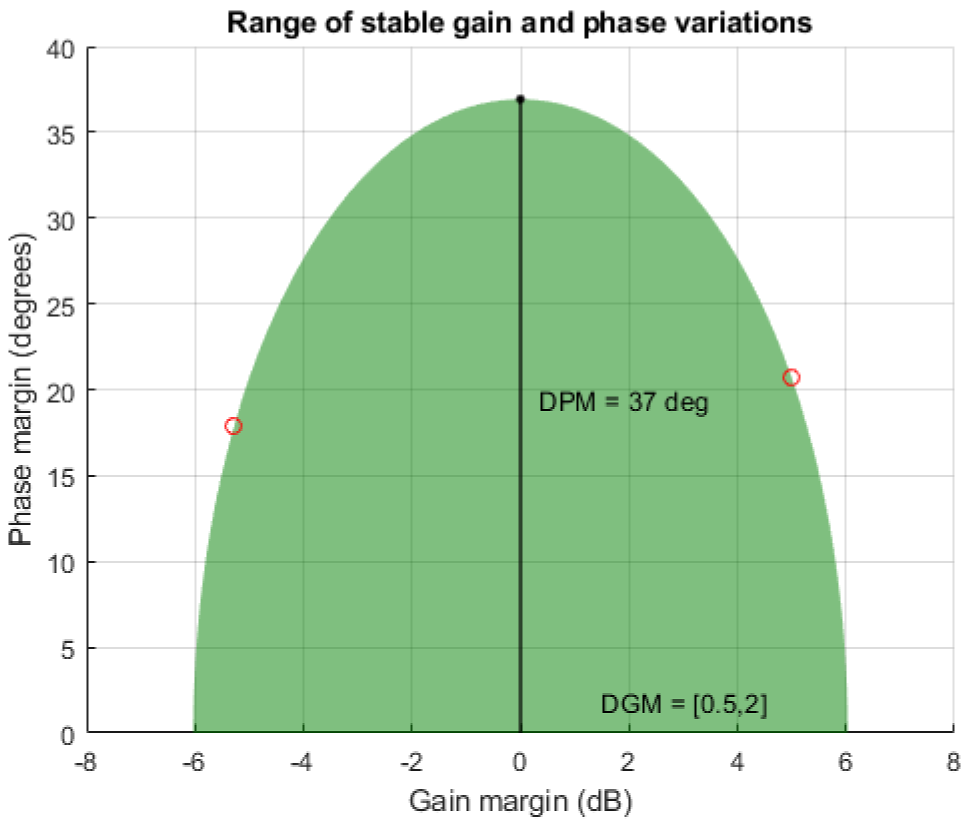
```
         Pole                   Damping       Frequency      Time Constant
                                            (rad/seconds)      (seconds)

 -1.88e-03                     1.00e+00       1.88e-03         5.33e+02
 -2.55e-02                     1.00e+00       2.55e-02         3.92e+01
 -3.20e-02                     1.00e+00       3.20e-02         3.12e+01
 -5.16e-02                     1.00e+00       5.16e-02         1.94e+01
 -1.25e-01                     1.00e+00       1.25e-01         7.98e+00
 -3.85e+00 + 5.04e+00i         6.07e-01       6.34e+00         2.60e-01
 -3.85e+00 - 5.04e+00i         6.07e-01       6.34e+00         2.60e-01
 -8.38e+00 + 1.17e+01i         5.81e-01       1.44e+01         1.19e-01
 -8.38e+00 - 1.17e+01i         5.81e-01       1.44e+01         1.19e-01
  1.28e-13 + 2.37e+01i        -5.42e-15       2.37e+01        -7.79e+12
  1.28e-13 - 2.37e+01i        -5.42e-15       2.37e+01        -7.79e+12
 -2.95e+01                     1.00e+00       2.95e+01         3.39e-02
 -3.33e+01                     1.00e+00       3.33e+01         3.00e-02
 -6.04e+01 + 2.28e+01i         9.36e-01       6.46e+01         1.66e-02
 -6.04e+01 - 2.28e+01i         9.36e-01       6.46e+01         1.66e-02
 -3.86e+01 + 5.36e+01i         5.85e-01       6.61e+01         2.59e-02
 -3.86e+01 - 5.36e+01i         5.85e-01       6.61e+01         2.59e-02
 -1.10e+03                     1.00e+00       1.10e+03         9.05e-04
```

```
w0 = MM.Frequency
```

```
w0 =

   23.6855
```

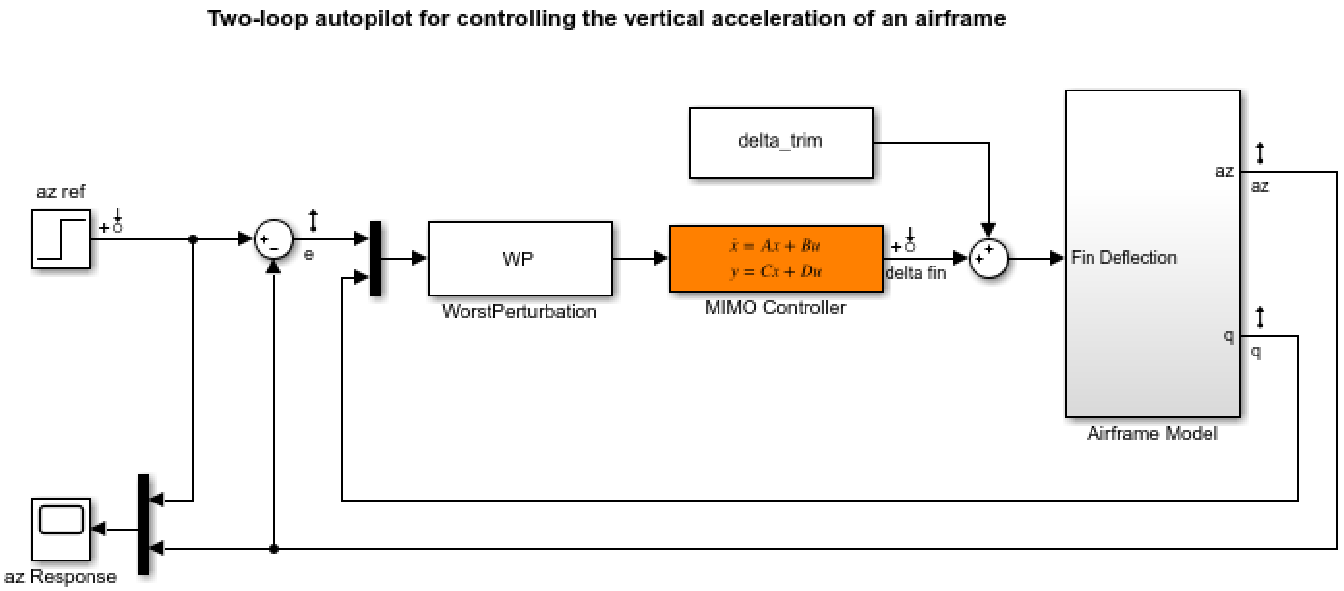
Note that the gain and phase variations induced by this perturbation lie on the boundary of the range of safe gain/phase variations computed by `diskmargin`. To confirm this result, compute the frequency response of the worst perturbation at the frequency `w0`, convert it to a gain and phase variation, and visualize it along with the range of safe gain and phase variations represented by the multiloop disk margin.

```
DELTA = freqresp(WP,w0);
clf
diskmarginplot(MM.GainMargin)
title('Range of stable gain and phase variations')
hold on
plot(20*log10(abs(DELTA(1,1))),abs( angle(DELTA(1,1))*180/pi),'ro')
plot(20*log10(abs(DELTA(2,2))),abs( angle(DELTA(2,2))*180/pi),'ro')
```



To simulate the effect of this worst perturbation on the full nonlinear model in Simulink, insert it as a block before the controller block, as done in the modified model `rct_airframeWP`.

```
close(f)
open_system('rct_airframeWP')
```



Here the MIMO Controller block is set to the nominal controller Knom. To simulate the nonlinear response with this controller, compute the trim deflection and q initial value from the operating condition op.

```
delta_trim = op.Inputs.u + [1.5 0]*op.States(5).x;
q_ini = op.States(4).x;
```

By commenting the WorstPerturbation block in and out, you can simulate the response with or without this perturbation. The results are shown below and confirm the destabilizing effect of the gain and phase variation computed by diskmargin.

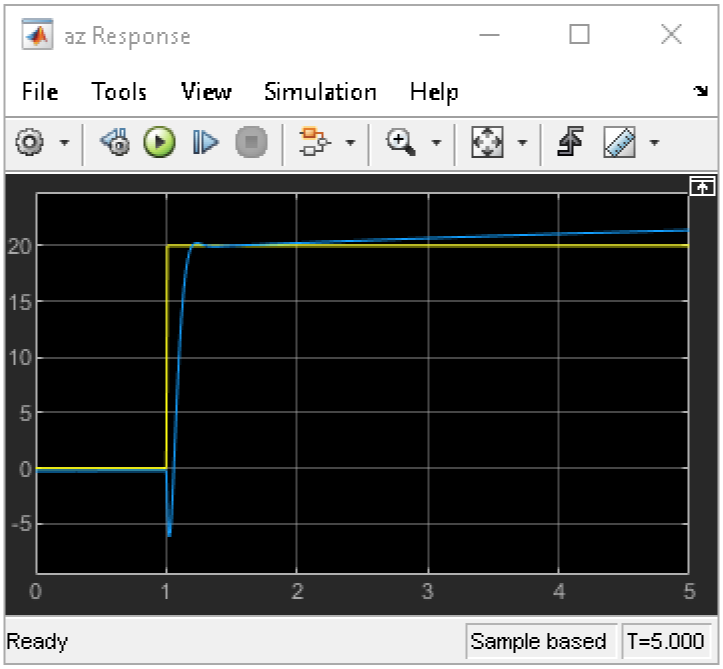


Figure 1: Nominal response.

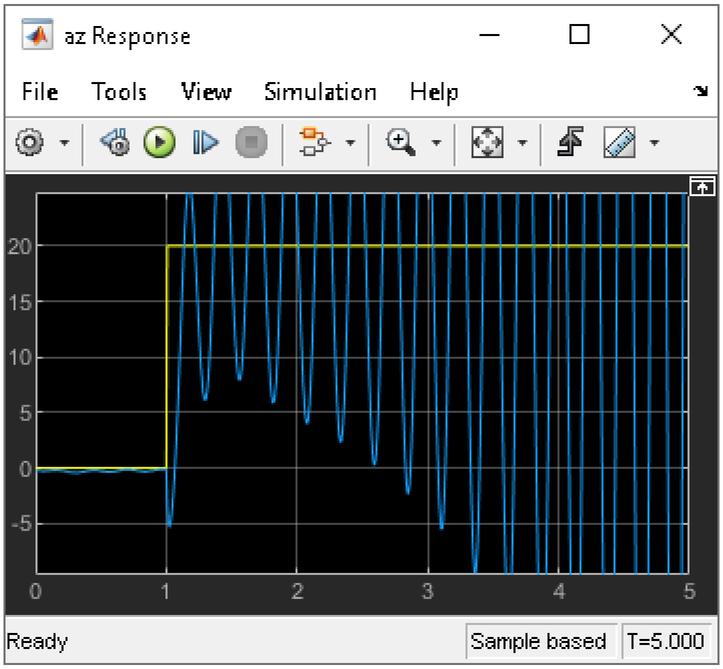


Figure 2: Response with destabilizing gain/phase perturbation.

See Also

`diskmargin` | `diskmarginplot` | `musyn` | `umargin`

More About

- “Stability Analysis Using Disk Margins” on page 2-2
- “Uncertain Gain and Phase” on page 1-12

Robust Controller for Spinning Satellite

This example expands on the “MIMO Stability Margins for Spinning Satellite” on page 2-14 example by designing a robust controller that overcomes the flaws of the "naive" design.

Plant model

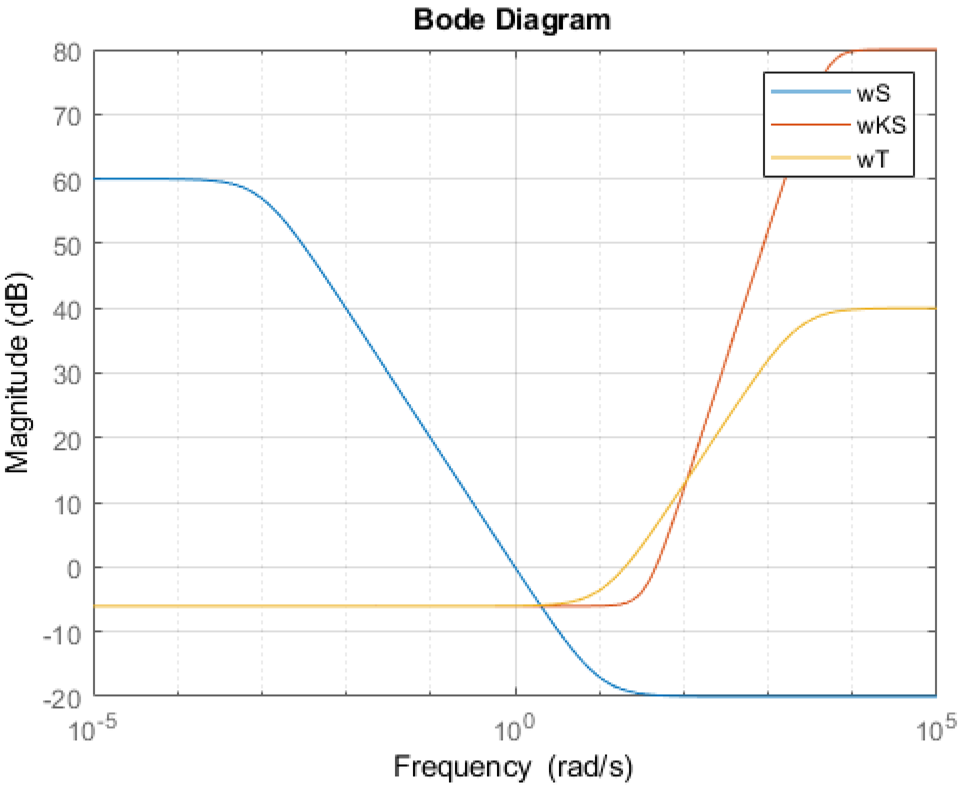
The plant model is the same as described in “MIMO Stability Margins for Spinning Satellite” on page 2-14.

```
a = 10;
A = [0 a;-a 0];
B = eye(2);
C = [1 a;-a 1];
D = 0;
Gnom = ss(A,B,C,D);
```

Nominal Mixed-Sensitivity Design

Start with a basic mixed-sensitivity design using `mixsyn`. Pick the weights to achieve good performance while limiting bandwidth and control effort. (See “Mixed-Sensitivity Loop Shaping” for details about this technique and how to choose weighting functions.)

```
wS = makeweight(1e3,1,1e-1);
wKS = makeweight(0.5,[500 1e2],1e4,0,2);
wT = makeweight(0.5,20,100);
bodemag(wS,wKS,wT), grid
legend('wS','wKS','wT')
```



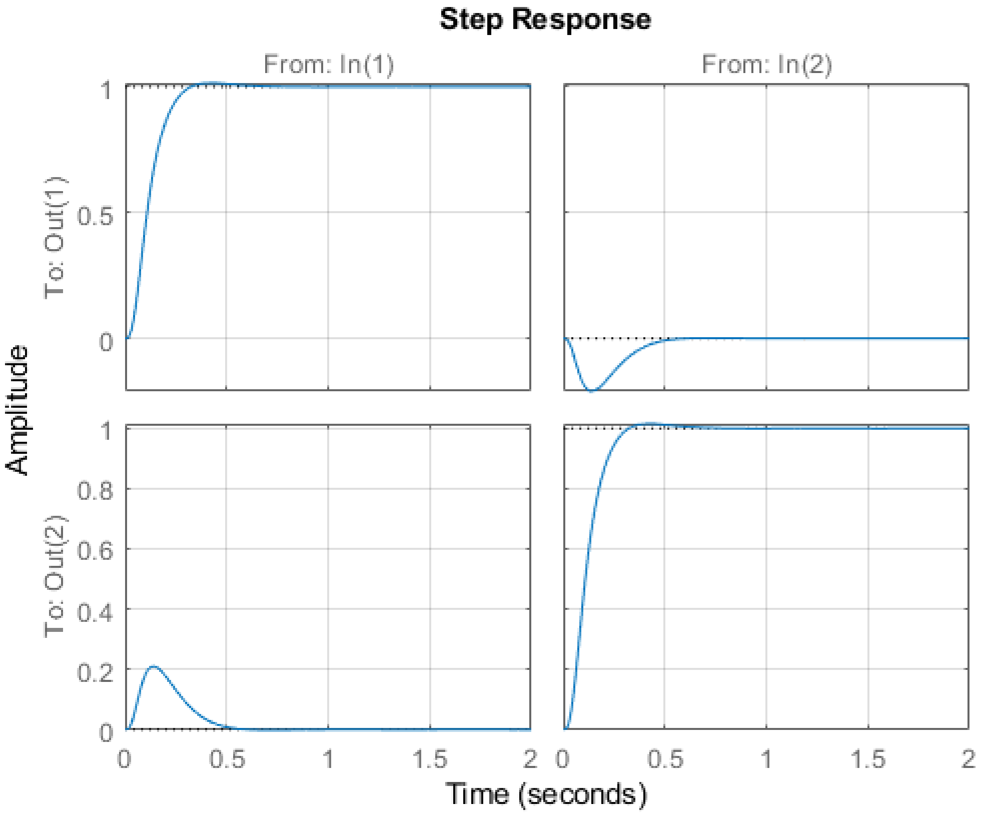
Compute the optimal MIMO controller `K1` with `mixsyn`.

```
[K1,~,gam] = mixsyn(Gnom,wS,wKS,wT);
gam
```

```
gam = 0.7166
```

The optimal performance is about 0.7, indicating that `mixsyn` easily met the bounds on S, KS, T. Close the feedback loop and plot the step response.

```
T = feedback(Gnom*K1,eye(2));
step(T,2), grid
```

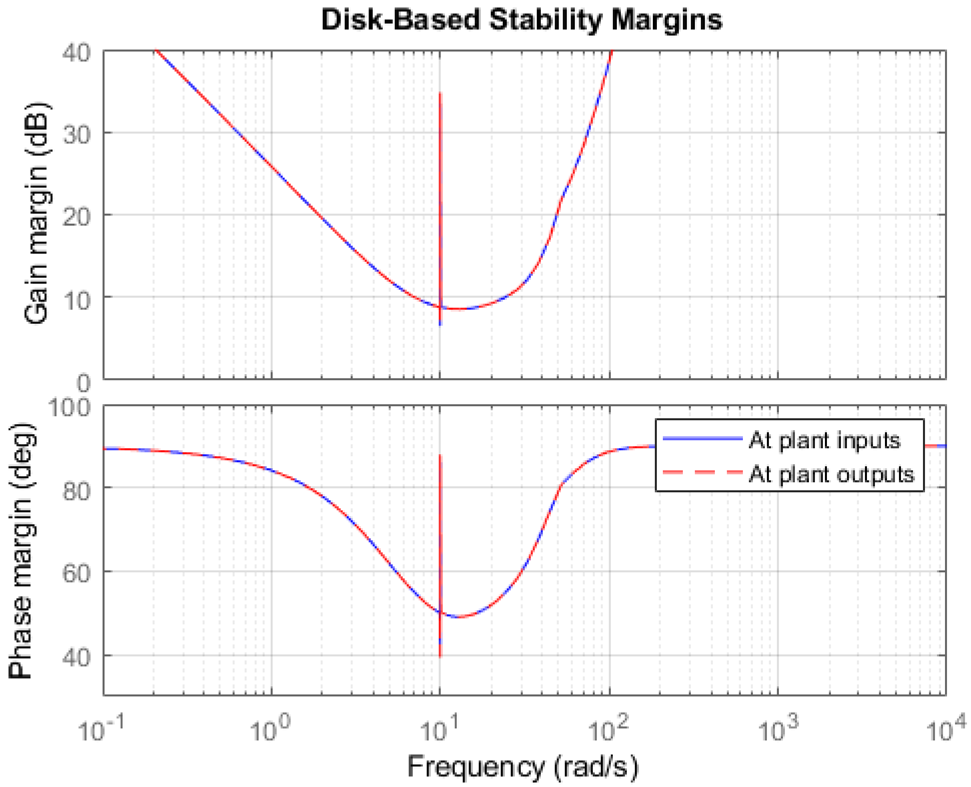


The nominal responses are fast with little overshoot.

Disk Margins

To gauge the robustness of this controller, check the disk margins at the plant inputs and the plant outputs.

```
diskmarginplot(K1*Gnom,'b',Gnom*K1,'r--')
grid
legend('At plant inputs','At plant outputs')
```



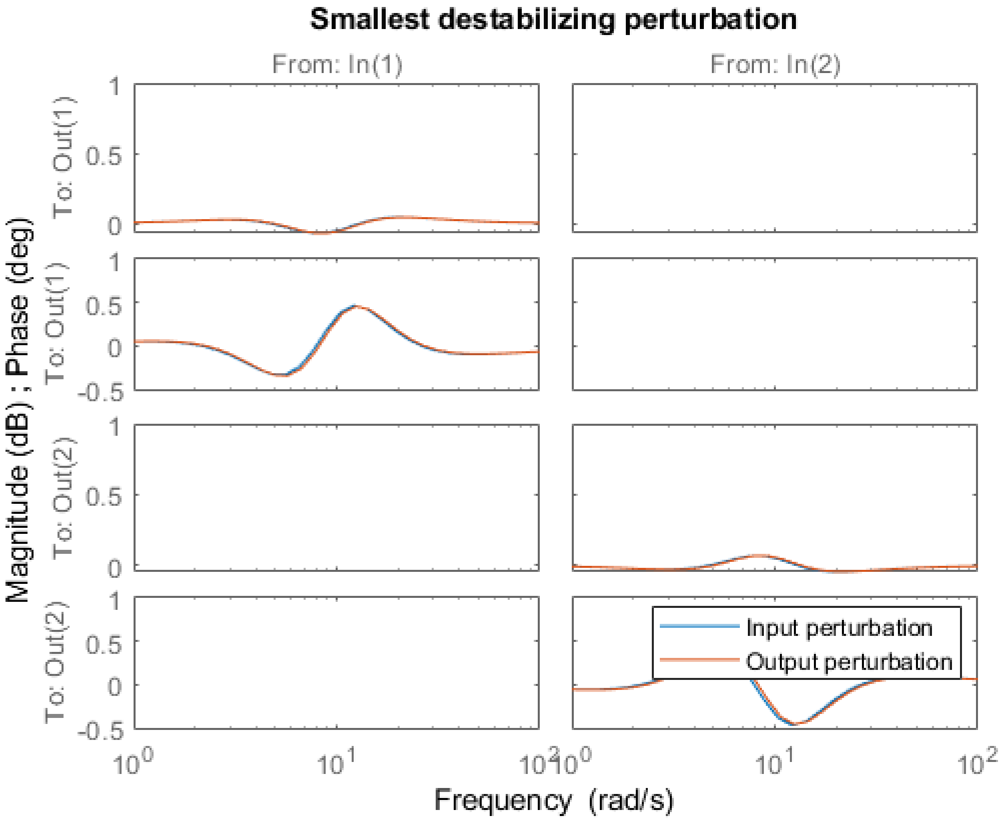
Both are good with near 10 dB gain margin and 50 degrees phase margin. Also check the disk margins when the gain and phase are allowed to vary at both the inputs and outputs of the plant.

```
MMIO = diskmargin(Gnom,K1)
```

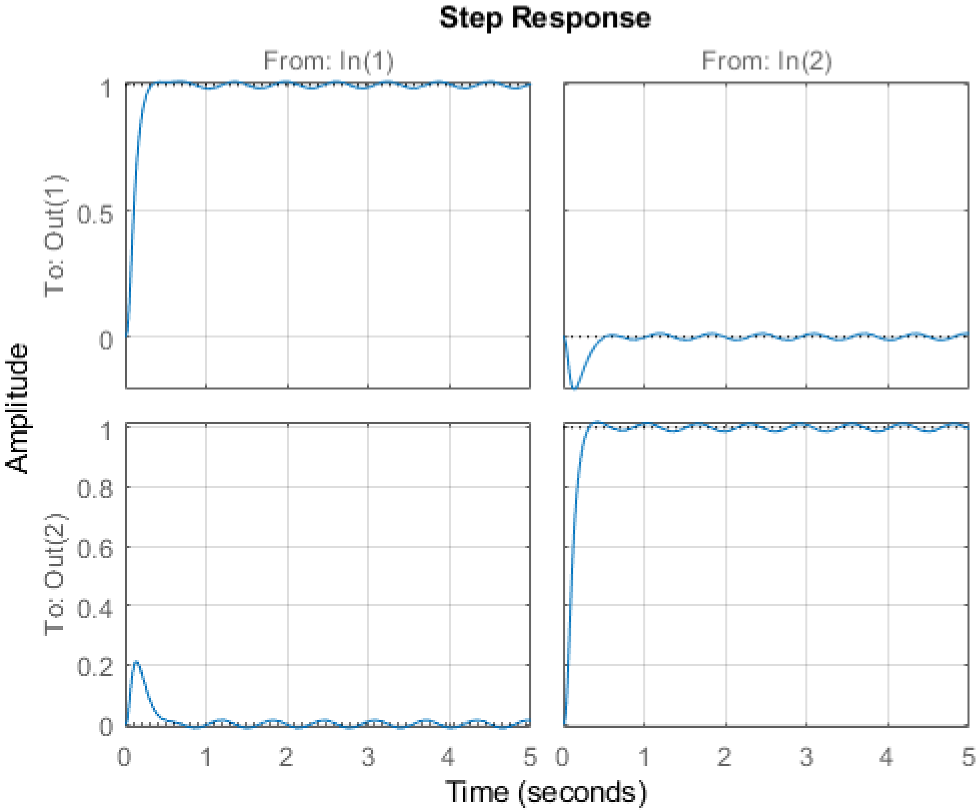
```
MMIO = struct with fields:
           GainMargin: [0.9915 1.0085]
          PhaseMargin: [-0.4863 0.4863]
           DiskMargin: 0.0085
           LowerBound: 0.0085
           UpperBound: 0.0085
            Frequency: 9.9988
    WorstPerturbation: [1x1 struct]
```

The I/O margins are extremely small. This first design also lacks robustness. You can confirm the poor robustness by injecting the smallest destabilizing perturbations returned by `diskmargin` at the plant inputs and outputs. (See `diskmargin` for further details about the `WorstPerturbation` field of its output structures.)

```
WP = MMIO.WorstPerturbation;
bode(WP.Input,WP.Output)
title('Smallest destabilizing perturbation')
legend('Input perturbation','Output perturbation')
```



```
Tpert = feedback(WP.Output*Gnom*WP.Input*K1,eye(2));
step(Tpert,5)
grid
```



The step response continues to oscillate after the initial transient indicating marginal instability. Verify that the perturbed closed-loop system `Tpert` has a pole on the imaginary axis at the critical frequency `MMIO.Frequency`.

```
[wn,zeta] = damp(Tpert);
[~,idx] = min(zeta);
[zeta(idx) wn(idx) MMIO.Frequency]
```

```
ans = 1×3

    0.0000    9.9988    9.9988
```

Robust Design

Create an uncertain plant model where the parameter `a` (spinning frequency) varies in the range [7 13].

```
a = ureal('a',10,'range',[7 13]);
A = [0 a;-a 0];
B = eye(2);
C = [1 a;-a 1];
D = 0;
Gunc = ss(A,B,C,D);
```

You can use `musyn` to design a robust controller for this uncertain plant. To improve robustness, use the `umargin` element to model gain and phase uncertainty at both inputs and outputs, so that musyn enforces robustness for the modeled range of uncertainty. Suppose that you want at least 2 dB gain margin at each I/O (4 dB total for each channel). If your umargin elements model that full range of variation, musyn might not yield good results, because it attempts to enforce robust performance over the modeled uncertainty as well as robust stability. musyn most likely cannot maintained the desired performance for that much gain variation. Instead, scale back the target to 1 dB gain variation in each I/O.

```
GM = 1.1; % about 1 dB
u1 = umargin('u1',GM);
u2 = umargin('u2',GM);
y1 = umargin('y1',GM);
y2 = umargin('y2',GM);
InputMargins = append(u1,u2);
OutputMargins = append(y1,y2);
Gunc = OutputMargins*Gunc*InputMargins;
```

Augment the plant with the mixed-sensitivity weights and use musyn to optimize robust performance for the modeled uncertainty, which includes both the parameter a and the gain and phase variations at plant inputs and outputs.

```
P = augw(Gunc,wS,wKS,wT);
[K2,gam] = musyn(P,2,2);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter         K Step       Peak MU       D Fit             D
    1           175.9        2.593        2.606            24
    2            1.21         1.21        1.226            58
    3            1.17         1.17        1.182            56
    4           1.168        1.168        1.176            64
    5           1.167        1.167        1.173            56

Best achieved robust performance: 1.17
```

The robust performance is close to 1, indicating that the controller is close to robustly meeting the mixed-sensitivity goals. Check the disk margins at the plant I/Os.

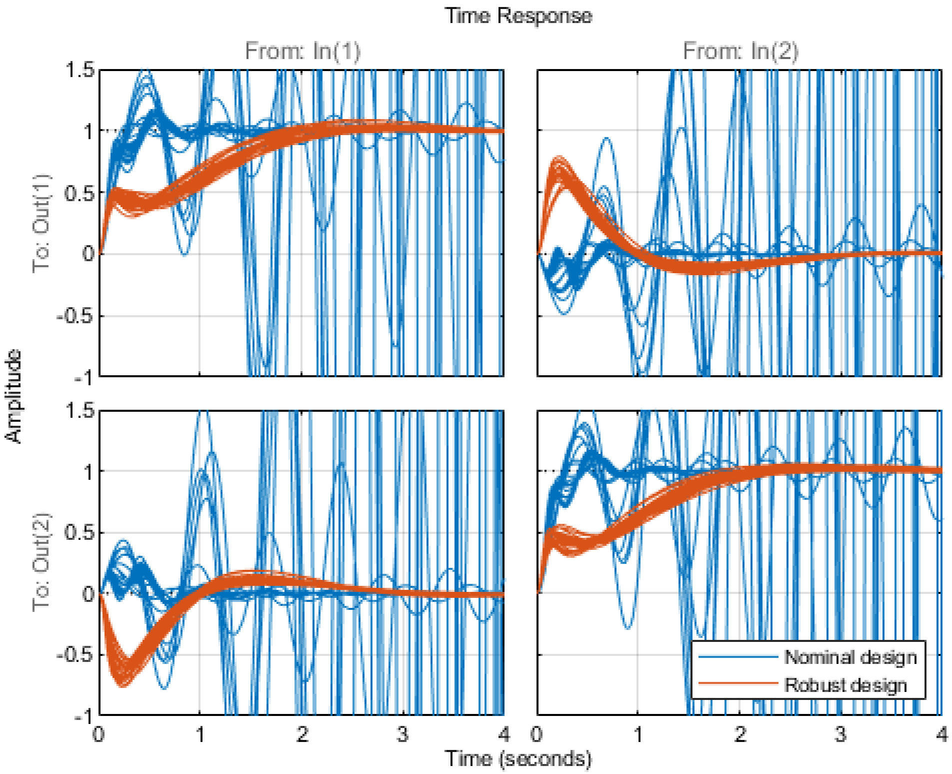
```
MMIO = diskmargin(Gnom,K2)
```

```
MMIO = struct with fields:
           GainMargin: [0.6396 1.5634]
          PhaseMargin: [-24.7926 24.7926]
           DiskMargin: 0.4396
           LowerBound: 0.4396
           UpperBound: 0.4487
            Frequency: 2.8513
    WorstPerturbation: [1x1 struct]
```

The margins are now about 1.6 dB and 25 degrees, much better than before. Compare the step responses with each controller for 25 uncertainty samples.

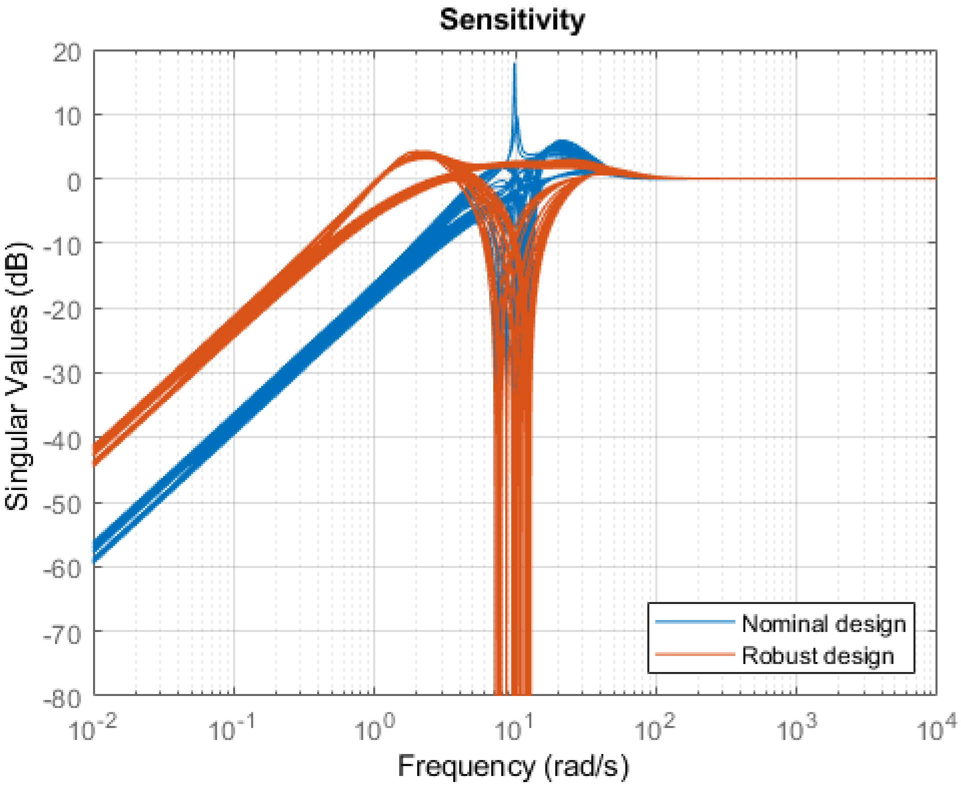
```
T1 = feedback(Gunc*K1,eye(2));
T2 = feedback(Gunc*K2,eye(2));
rng(0) % for reproducibility
T1s = usample(T1,25);
```

```
rng(0)
T2s = usample(T2,25);
opt = timeoptions;
opt.YLim = {[-1 1.5]};
stepplot(T1s,T2s,4,opt)
grid
legend('Nominal design','Robust design','location','southeast')
```

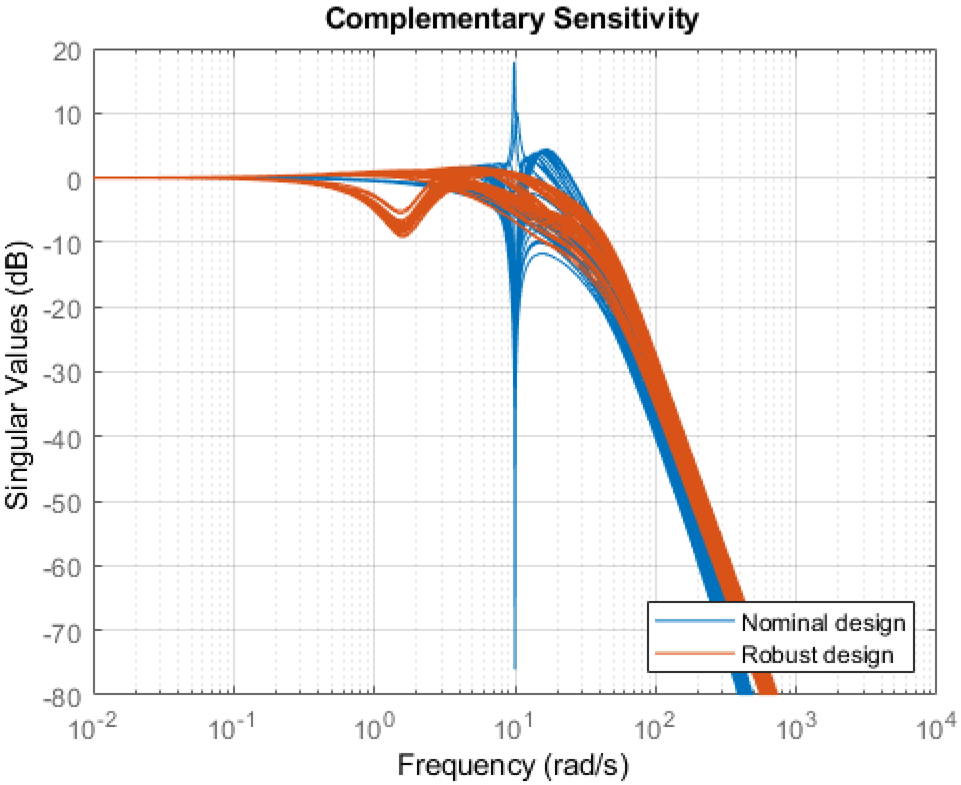


The second design is a clear improvement. Further compare the sensitivity and complementary sensitivity functions.

```
sigma(eye(2)-T1s,eye(2)-T2s), grid
axis([1e-2 1e4 -80 20])
title('Sensitivity')
legend('Nominal design','Robust design','location','southeast')
```



```
sigma(T1s,T2s), grid
axis([1e-2 1e4 -80 20])
title('Complementary Sensitivity')
legend('Nominal design','Robust design','location','southeast')
```



This example has shown how to use the `umargin` uncertain element to improve stability margins as part of a robust controller synthesis.

See Also

`diskmargin` | `musyn` | `umargin`

More About

- “Stability Analysis Using Disk Margins” on page 2-2
- “Uncertain Gain and Phase” on page 1-12

Introduction to Linear Matrix Inequalities

Linear Matrix Inequalities

Linear Matrix Inequalities (LMIs) and LMI techniques have emerged as powerful design tools in areas ranging from control engineering to system identification and structural design. Three factors make LMI techniques appealing:

- A variety of design specifications and constraints can be expressed as LMIs.
- Once formulated in terms of LMIs, a problem can be solved *exactly* by efficient convex optimization algorithms (see "LMI Solvers" on page 5-18).
- While most problems with multiple constraints or objectives lack analytical solutions in terms of matrix equations, they often remain tractable in the LMI framework. This makes LMI-based design a valuable alternative to classical "analytical" methods.

See [9] for a good introduction to LMI concepts. Robust Control Toolbox software is designed as an easy and progressive gateway to the new and fast-growing field of LMIs:

- For users who occasionally need to solve LMI problems, the LMI Editor and the tutorial introduction to LMI concepts and LMI solvers provide for quick and easy problem solving.
- For more experienced LMI users, LMI Lab, offers a rich, flexible, and fully programmable environment to develop customized LMI-based tools.

LMI Features

Robust Control Toolbox LMI functionality serves two purposes:

- Provide state-of-the-art tools for the LMI-based analysis and design of robust control systems
- Offer a flexible and user-friendly environment to specify and solve general LMI problems (the LMI Lab)

Examples of LMI-based analysis and design tools include

- Functions to analyze the robust stability and performance of uncertain systems with varying parameters (`popov`, `quadstab`, `quadperf` ...)
- Functions to design robust control with a mix of H_2, H_∞, and pole placement objectives (`h2hinfsyn`)
- Functions for synthesizing robust gain-scheduled H_∞ controllers (`hinfgs`)

For users interested in developing their own applications, the LMI Lab provides a general-purpose and fully programmable environment to specify and solve virtually any LMI problem. Note that the scope of this facility is by no means restricted to control-oriented applications.

Note Robust Control Toolbox software implements state-of-the-art interior-point LMI solvers. While these solvers are significantly faster than classical convex optimization algorithms, you should keep in mind that the complexity of LMI computations can grow quickly with the problem order (number of states). For example, the number of operations required to solve a Riccati equation is $o(n^3)$ where *n* is the state dimension, while the cost of solving an equivalent "Riccati inequality" LMI is $o(n^6)$.

See Also

More About

- “LMIs and LMI Problems” on page 4-4

LMIs and LMI Problems

A linear matrix inequality (LMI) is any constraint of the form

$$A(x) := A_0 + x_1A_1 + \ldots + x_NA_N < 0 \tag{4-1}$$

where

- $x = (x_1, \ldots, x_N)$ is a vector of unknown scalars (the *decision* or *optimization* variables)
- $A_0, \ldots, A_N$ are given *symmetric* matrices
- < 0 stands for "negative definite," i.e., the largest eigenvalue of $A(x)$ is negative

Note that the constraints $A(x) > 0$ and $A(x) < B(x)$ are special cases of "Equation 4-1" since they can be rewritten as $-A(x) < 0$ and $A(x) - B(x) < 0$, respectively.

The LMI of "Equation 4-1" is a convex constraint on x since $A(y) < 0$ and $A(z) < 0$ imply that $A\left(\frac{y+z}{2}\right) < 0$. As a result,

- Its solution set, called the *feasible set*, is a convex subset of R^N
- Finding a solution x to "Equation 4-1", if any, is a convex optimization problem.

Convexity has an important consequence: even though "Equation 4-1" has no analytical solution in general, it can be solved numerically with guarantees of finding a solution when one exists. Note that a system of LMI constraints can be regarded as a single LMI since

$$\begin{cases} A_1(x) < 0 \\ \quad\vdots \\ A_K(x) < 0 \end{cases}$$

is equivalent to

$$A(x) := \mathrm{diag}(\mathrm{A}_1(x), \ldots, \mathrm{A}_\mathrm{K}(x)) < 0$$

where diag $(A_1(x), \ldots, A_\mathrm{K}(x))$ denotes the block-diagonal matrix with $A_1(x), \ldots, A_\mathrm{K}(x)$ on its diagonal. Hence multiple LMI constraints can be imposed on the vector of decision variables x without destroying convexity.

In most control applications, LMIs do not naturally arise in the canonical form of "Equation 4-1" , but rather in the form

$$L(X_1, \ldots, X_\mathrm{n}) < R(X_1, \ldots, X_\mathrm{n})$$

where $L(.)$ and $R(.)$ are affine functions of some structured *matrix* variables $X_1, \ldots, X_\mathrm{n}$. *A* simple example is the Lyapunov inequality

$$A^TX + XA < 0 \tag{4-2}$$

where the unknown X is a symmetric matrix. Defining $x_1, \ldots, x_\mathrm{N}$ as the independent scalar entries of X, this LMI could be rewritten in the form of "Equation 4-1". Yet it is more convenient and efficient to describe it in its natural form "Equation 4-2", which is the approach taken in the LMI Lab.

See Also

Related Examples

- “LMI Applications” on page 4-6

LMI Applications

Finding a solution x to the LMI system

$$A(x) < 0 \tag{4-3}$$

is called the feasibility problem. Minimizing a convex objective under LMI constraints is also a convex problem. In particular, the *linear objective minimization problem*:

Minimize $c^T x$ subject to

$$A(x) < 0 \tag{4-4}$$

plays an important role in LMI-based design. Finally, the *generalized eigenvalue minimization problem*

Minimize λ subject to

$$\begin{gathered} A(x) < \lambda B(x) \\ B(x) > 0 \\ C(x) > 0 \end{gathered} \tag{4-5}$$

is quasi-convex and can be solved by similar techniques. It owes its name to the fact that is related to the largest generalized eigenvalue of the pencil $(A(x),B(x))$.

Many control problems and design specifications have LMI formulations [9]. This is especially true for Lyapunov-based analysis and design, but also for optimal LQG control, H^∞ control, covariance control, etc. Further applications of LMIs arise in estimation, identification, optimal design, structural design [6], [7], matrix scaling problems, and so on. The main strength of LMI formulations is the ability to combine various design constraints or objectives in a numerically tractable manner.

A nonexhaustive list of problems addressed by LMI techniques includes the following:

- Robust stability of systems with LTI uncertainty (µ-analysis) ([24], [21], [27])
- Robust stability in the face of sector-bounded nonlinearities (Popov criterion) ([22], [28], [13], [16])
- Quadratic stability of differential inclusions ([15], [8])
- Lyapunov stability of parameter-dependent systems ([12])
- Input/state/output properties of LTI systems (invariant ellipsoids, decay rate, etc.) ([9])
- Multi-model/multi-objective state feedback design ([4], [17], [3], [9], [10])
- Robust pole placement
- Optimal LQG control ([9])
- Robust H^∞ control ([11], [14])
- Multi-objective H^∞ synthesis ([18], [23], [10], [18])
- Design of robust gain-scheduled controllers ([5], [2])
- Control of stochastic systems ([9])
- Weighted interpolation problems ([9])

To hint at the principles underlying LMI design, let's review the LMI formulations of a few typical design objectives.

Stability

The stability of the dynamic system

$$\dot{x} = Ax$$

is equivalent to the feasibility of the following problem:

Find $P = P^T$ such that $A^T P + P A < 0$, $P > I$.

This can be generalized to linear differential inclusions (LDI)

$$\dot{x} = A(t)x$$

where $A(t)$ varies in the convex envelope of a set of LTI models:

$$A(t) \in \mathrm{Co}\{A_1, \ldots, A_n\} = \left\{ \sum_{i=1}^{n} a_i A_i : a_i \geq 0, \sum_{i=1}^{N} a_i = 1 \right\}.$$

A sufficient condition for the asymptotic stability of this LDI is the feasibility of

Find $P = P^T$ such that $A_i^T P + P A_i < 0, \quad P > I$.

RMS Gain

The random-mean-squares (RMS) gain of a stable LTI system

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

is the largest input/output gain over all bounded inputs $u(t)$. This gain is the global minimum of the following linear objective minimization problem [1], [25], [26].

Minimize γ over $X = X^T$ and γ such that

$$\begin{pmatrix} A^T X + XA & XB & C^T \\ B^T X & -\gamma I & D^T \\ C & D & -\gamma I \end{pmatrix} < 0$$

and

$$X > 0.$$

LQG Performance

For a stable LTI system

$$G \begin{cases} \dot{x} = Ax + Bw \\ y = Cx \end{cases}$$

where w is a white noise disturbance with unit covariance, the LQG or H_2 performance $\|G\|_2$ is defined by

$$\|G\|_2^2:\ =\lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T y^T(t)y(t)dt\right\}$$

$$=\frac{1}{2\pi}\int_{-\infty}^{\infty} G^H(j\omega)G(j\omega)d\omega\,.$$

It can be shown that

$$\|G\|_2^2=\inf\left\{\mathrm{Trace}\left(\mathrm{CPC}^T\right):AP+PA^T+BB^T<0\right\}.$$

Hence $\|G\|_2^2$ is the global minimum of the LMI problem. Minimize Trace (Q) over the symmetric matrices P,Q such that

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

and

$$\begin{pmatrix} Q & CP \\ PC^T & P \end{pmatrix}>0.$$

Again this is a linear objective minimization problem since the objective Trace (Q) is linear in the decision variables (free entries of P,Q).

See Also

More About

- “Tools for Specifying and Solving LMIs” on page 5-2

Further Mathematical Background

Efficient interior-point algorithms are now available to solve the three generic LMI problems “Equation 4-2”–“Equation 4-4” defined in “LMI Applications” on page 4-6. These algorithms have a polynomial-time complexity. That is, the number $N(\varepsilon)$ of flops needed to compute an ε-accurate solution is bounded by

$$M\, N^3 \log(V/\varepsilon)$$

where M is the total row size of the LMI system, N is the total number of scalar decision variables, and V is a data-dependent scaling factor. Robust Control Toolbox software implements the Projective Algorithm of Nesterov and Nemirovski [20], [19]. In addition to its polynomial-time complexity, this algorithm does not require an initial feasible point for the linear objective minimization problem “Equation 4-3”or the generalized eigenvalue minimization problem “Equation 4-4”.

Some LMI problems are formulated in terms of inequalities rather than strict inequalities. For instance, a variant of “Equation 4-3” is

Minimize $c^T x$ subject to $A(x) < 0$.

While this distinction is immaterial in general, it matters when $A(x)$ can be made negative semi-definite but not negative definite. A simple example is:

Minimize $c^T x$ subject to

$$\begin{pmatrix} x & x \\ x & x \end{pmatrix} \geq 0. \tag{4-6}$$

Such problems cannot be handled directly by interior-point methods which require strict feasibility of the LMI constraints. A well-posed reformulation of “Equation 4-5” would be

Minimize $c^T x$ subject to $x \geq 0$.

Keeping this subtlety in mind, we always use strict inequalities in this manual.

See Also

Related Examples

- “LMI Applications” on page 4-6

Bibliography

[1] Anderson, B.D.O., and S. Vongpanitlerd, Network Analysis, Prentice-Hall, Englewood Cliffs, 1973.

[2] Apkarian, P., P. Gahinet, and G. Becker, “Self-Scheduled H^∞ Control of Linear Parameter-Varying Systems,” *Proc. Amer. Contr. Conf.*, 1994, pp. 856-860.

[3] Bambang, R., E. Shimemura, and K. Uchida, “Mixed H_2 /H^∞ Control with Pole Placement,” State-Feedback Case, *Proc. Amer. Contr. Conf.*, 1993, pp. 2777-2779.

[4] Barmish, B.R., “Stabilization of Uncertain Systems via Linear Control,” *IEEE Trans. Aut. Contr.*, AC–28 (1983), pp. 848-850.

[5] Becker, G., and Packard, P., “Robust Performance of Linear-Parametrically Varying Systems Using Parametrically-Dependent Linear Feedback,” *Systems and Control Letters*, 23 (1994), pp. 205-215.

[6] Bendsoe, M.P., A. Ben-Tal, and J. Zowe, “Optimization Methods for Truss Geometry and Topology Design,” to appear in *Structural Optimization*.

[7] Ben-Tal, A., and A. Nemirovski, “Potential Reduction Polynomial-Time Method for Truss Topology Design,” to appear in *SIAM J. Contr. Opt*.

[8] Boyd, S., and Q. Yang, “Structured and Simultaneous Lyapunov Functions for System Stability Problems,” *Int. J. Contr.*, 49 (1989), pp. 2215-2240.

[9] Boyd, S., L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in Systems and Control Theory*, SIAM books, Philadelphia, 1994.

[10] Chilali, M., and P. Gahinet, “H^∞ Design with Pole Placement Constraints: an LMI Approach,” to appear in *IEEE Trans. Aut. Contr*. Also in *Proc. Conf. Dec. Contr.*, 1994, pp. 553-558.

[11] Gahinet, P., and P. Apkarian, “A Linear Matrix Inequality Approach to H^∞ Control,” *Int. J. Robust and Nonlinear Contr.*, 4 (1994), pp. 421-448.

[12] Gahinet, P., P. Apkarian, and M. Chilali, “Affine Parameter-Dependent Lyapunov Functions for Real Parametric Uncertainty,“ *Proc. Conf. Dec. Contr.*, 1994, pp. 2026-2031.

[13] Haddad, W.M., and D.S. Berstein, “Parameter-Dependent Lyapunov Functions, Constant Real Parameter Uncertainty, and the Popov Criterion in Robust Analysis and Synthesis: Part 1 and 2,” *Proc. Conf. Dec. Contr.*, 1991, pp. 2274-2279 and 2632-2633.

[14] Iwasaki, T., and R.E. Skelton, “All Controllers for the General H^∞ Control Problem: LMI Existence Conditions and State-Space Formulas,” *Automatica*, 30 (1994), pp. 1307-1317.

[15] Horisberger, H.P., and P.R. Belanger, “Regulators for Linear Time-Varying Plants with Uncertain Parameters,” *IEEE Trans. Aut. Contr.*, AC–21 (1976), pp. 705-708.

[16] How, J.P., and S.R. Hall, “Connection between the Popov Stability Criterion and Bounds for Real Parameter Uncertainty,” *Proc. Amer. Contr. Conf.*, 1993, pp. 1084-1089.

[17] Khargonekar, P.P., and M.A. Rotea, “Mixed H_2 /H^∞ Control: a Convex Optimization Approach,” *IEEE Trans. Aut. Contr.*, 39 (1991), pp. 824-837.

[18] Masubuchi, I., A. Ohara, and N. Suda, “LMI-Based Controller Synthesis: *A* Unified Formulation and Solution,” submitted to *Int. J. Robust and Nonlinear Contr.*, 1994.

[19] Nemirovski, A., and P. Gahinet, “The Projective Method for Solving Linear Matrix Inequalities,” *Proc. Amer. Contr. Conf.*, 1994, pp. 840-844.

[20] Nesterov, Yu, and A. Nemirovski, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, SIAM Books, Philadelphia, 1994.

[21] Packard, A., and J.C. Doyle, “The Complex Structured Singular Value,” *Automatica*, 29 (1994), pp. 71-109.

[22] Popov, V.M., “Absolute Stability of Nonlinear Systems of Automatic Control,” *Automation and Remote Control*, 22 (1962), pp. 857-875.

[23] Scherer, C., “Mixed H_2 H^∞ Control,” to appear in *Trends in Control: A European Perspective,* volume of the special contributions to the ECC 1995.

[24] Stein, G., and J.C. Doyle, “Beyond Singular Values and Loop Shapes,” *J. Guidance*, 14 (1991), pp. 5-16.

[25] Vidyasagar, M., *Nonlinear System Analysis*, Prentice-Hall, Englewood Cliffs, 1992.

[26] Willems, J.C., “Least-Squares Stationary Optimal Control and the Algebraic Riccati Equation,” *IEEE Trans. Aut. Contr.*, AC–16 (1971), pp. 621-634.

[27] Young, P.M., M.P. Newlin, and J.C. Doyle, “Let's Get Real,” in *Robust Control Theory*, Springer Verlag, 1994, pp. 143-174.

[28] Zames, G., “On the Input-Output Stability of Time-Varying Nonlinear Feedback Systems, Part I and II,” *IEEE Trans. Aut. Contr.*, AC–11 (1966), pp. 228-238 and 465-476.

LMI Lab

Tools for Specifying and Solving LMIs

The LMI Lab is a high-performance package for solving general LMI problems. It blends simple tools for the specification and manipulation of LMIs with powerful LMI solvers for three generic LMI problems. Thanks to a structure-oriented representation of LMIs, the various LMI constraints can be described in their natural block-matrix form. Similarly, the optimization variables are specified directly as *matrix variables* with some given structure. Once an LMI problem is specified, it can be solved numerically by calling the appropriate LMI solver. The three solvers `feasp`, `mincx`, and `gevp` constitute the computational engine of the LMI portion of Robust Control Toolbox software. Their high performance is achieved through C-MEX implementation and by taking advantage of the particular structure of each LMI.

The LMI Lab offers tools to

- Specify LMI systems either symbolically with the LMI Editor or incrementally with the `lmivar` and `lmiterm` commands
- Retrieve information about existing systems of LMIs
- Modify existing systems of LMIs
- Solve the three generic LMI problems (feasibility problem, linear objective minimization, and generalized eigenvalue minimization)
- Validate results

This chapter gives a tutorial introduction to the LMI Lab as well as more advanced tips for making the most out of its potential.

Some Terminology

Any linear matrix inequality can be expressed in the canonical form

$$L(x) = L_0 + x_1 L_1 + . . . + x_N L_N < 0$$

where

- $L_0, L_1, . . . , L_N$ are given symmetric matrices
- $x = (x_1, . . . , x_N)^T \epsilon R^N$ is the vector of scalar variables to be determined. We refer to $x_1, . . . , x_N$ as the *decision variables*. The names “design variables” and “optimization variables” are also found in the literature.

Even though this canonical expression is generic, LMIs rarely arise in this form in control applications. Consider for instance the Lyapunov inequality

$$A^T X + XA < 0 \qquad (5\text{-}1)$$

where

$$A = \begin{pmatrix} -1 & 2 \\ 0 & -2 \end{pmatrix}$$

and the variable

$$X = \begin{pmatrix} x_1 & x_2 \\ x_2 & x_3 \end{pmatrix}$$

is a symmetric matrix. Here the decision variables are the free entries x_1, x_2, x_3 of X and the canonical form of this LMI reads

$$x_1\begin{pmatrix}-2 & 2\\ 2 & 0\end{pmatrix} + x_2\begin{pmatrix}0 & -3\\ -3 & 4\end{pmatrix} + x_3\begin{pmatrix}0 & 0\\ 0 & -4\end{pmatrix} < 0. \tag{5-2}$$

Clearly this expression is less intuitive and transparent than “Equation 5-1”. Moreover, the number of matrices involved in “Equation 5-2” grows roughly as n^2 /2 if n is the size of the A matrix. Hence, the canonical form is very inefficient from a storage viewpoint since it requires storing $o(n^2/2)$ matrices of size n when the single n-by-n matrix A would be sufficient. Finally, working with the canonical form is also detrimental to the efficiency of the LMI solvers. For these various reasons, the LMI Lab uses a *structured representation* of LMIs. For instance, the expression $A^TX + XA$ in the Lyapunov inequality “Equation 5-1” is explicitly described as a function of the matrix variable X, and only the A matrix is stored.

In general, LMIs assume a block matrix form where each block is an affine combination of the matrix variables. As a fairly typical illustration, consider the following LMI drawn from H^∞ theory

$$N^T\begin{pmatrix}A^TX + XA & XC^T & B\\ CX & -\gamma I & D\\ B^T & D^T & -\gamma I\end{pmatrix}N < 0 \tag{5-3}$$

where A, B, C, D, and N are given matrices and the problem variables are $X = X^T \epsilon R^{n\times n}$ and $\gamma \epsilon R$. We use the following terminology to describe such LMIs:

- N is called the *outer factor*, and the block matrix

  $$L(X,\gamma) = \begin{pmatrix}A^TX + XA & XC^T & B\\ CX & -\gamma I & D\\ B^T & D^T & -\gamma I\end{pmatrix}$$

  is called the *inner factor*. The outer factor *needs not be square* and is *often absent*.
- X and γ are the *matrix variables* of the problem. Note that scalars are considered as 1-by-1 matrices.
- The inner factor $L(X, \gamma)$ is a symmetric *block matrix*, its block structure being characterized by the sizes of its diagonal blocks. By symmetry, $L(X, \gamma)$ is entirely specified by the blocks on or above the diagonal.
- Each block of $L(X, \gamma)$ is an affine expression in the matrix variables X and γ. This expression can be broken down into a sum of elementary *terms*. For instance, the block (1,1) contains two elementary terms: A^TX and XA.
- Terms are either *constant* or *variable*. Constant terms are fixed matrices like B and D above. Variable terms involve one of the matrix variables, like XA, XC^T, and $-\gamma I$ above.

The LMI (“Equation 5-3”) is specified by the list of terms in each block, as is any LMI regardless of its complexity.

As for the matrix variables X and γ, they are characterized by their dimensions and structure. Common structures include rectangular unstructured, symmetric, skew-symmetric, and scalar. More sophisticated structures are sometimes encountered in control problems. For instance, the matrix variable X could be constrained to the block-diagonal structure:

$$X = \begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_2 & x_3 \\ 0 & x_3 & x_4 \end{pmatrix}.$$

Another possibility is the symmetric Toeplitz structure:

$$X = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_2 & x_1 & x_2 \\ x_3 & x_2 & x_1 \end{pmatrix}.$$

Summing up, structured LMI problems are specified by declaring the matrix variables and describing the term content of each LMI. This term-oriented description is systematic and accurately reflects the specific structure of the LMI constraints. There is no built-in limitation on the number of LMIs that you can specify or on the number of blocks and terms in any given LMI. LMI systems of arbitrary complexity can therefore, be defined in the LMI Lab.

Overview of the LMI Lab

The LMI Lab offers tools to specify, manipulate, and numerically solve LMIs. Its main purpose is to

- Allow for straightforward description of LMIs in their natural block-matrix form
- Provide easy access to the LMI solvers (optimization codes)
- Facilitate result validation and problem modification

The structure-oriented description of a given LMI system is stored as a single vector called the *internal representation* and generically denoted by `LMISYS` in the sequel. This vector encodes the structure and dimensions of the LMIs and matrix variables, a description of all LMI terms, and the related numerical data. It must be stressed that you need not attempt to read or understand the content of `LMISYS` since all manipulations involving this internal representation can be performed in a transparent manner with LMI-Lab tools.

The LMI Lab supports the following functionalities:

Specification of a System of LMIs

LMI systems can be either specified as symbolic matrix expressions with the interactive graphical user interface `lmiedit`, or assembled incrementally with the two commands `lmivar` and `lmiterm`. The first option is more intuitive and transparent while the second option is more powerful and flexible.

Information Retrieval

The interactive function `lmiinfo` answers qualitative queries about LMI systems created with `lmiedit` or `lmivar` and `lmiterm`. You can also use `lmiedit` to visualize the LMI system produced by a particular sequence of `lmivar`/`lmiterm` commands.

Solvers for LMI Optimization Problems

General-purpose LMI solvers are provided for the three generic LMI problems defined in “LMI Applications” on page 4-6. These solvers can handle very general LMI systems and matrix variable structures. They return a feasible or optimal vector of decision variables x^*. The corresponding values $X_1^*, \ldots, X_K^*$ of the matrix variables are given by the function `dec2mat`.

Result Validation

The solution *x** produced by the LMI solvers is easily validated with the functions `evallmi` and `showlmi`. This allows a fast check and/or analysis of the results. With `evallmi`, all variable terms in the LMI system are evaluated for the value *x** of the decision variables. The left and right sides of each LMI then become constant matrices that can be displayed with `showlmi`.

Modification of a System of LMIs

An existing system of LMIs can be modified in two ways:

- An LMI can be removed from the system with `dellmi`.
- A matrix variable *X* can be deleted using `delmvar`. It can also be instantiated, that is, set to some given matrix value. This operation is performed by `setmvar` and allows, for example, to fix some variables and solve the LMI problem with respect to the remaining ones.

See Also

Related Examples

- “Specifying a System of LMIs” on page 5-6

Specifying a System of LMIs

The LMI Lab can handle any system of LMIs of the form

$N^T L(X_1, . . . , X_K) N < MT R(X_1, . . . , X_K) M$

where

- $X_1, . . . , X_K$ are matrix variables with some prescribed structure
- The left and right outer factors N and M are given matrices with *identical* dimensions
- The left and right inner factors $L(.)$ and $R(.)$ are symmetric block matrices with identical block structures, each block being an affine combination of $X_1, . . . , X_K$ and their transposes.

Note Throughout this chapter, “left side” refers to what is on the “smaller” side of the inequality, and “right side” to what is on the “larger” side. Accordingly, X is called the right-hand side and 0 the left side of the LMI

$0 < X$

even when this LMI is written as $X > 0$.

The specification of an LMI system involves two steps:

1. Declare the dimensions and structure of each matrix variable $X_1, . . . , X_K$.
2. Describe the term content of each LMI.

This process creates the so-called *internal representation* of the LMI system. This computer description of the problem is used by the LMI solvers and in all subsequent manipulations of the LMI system. It is stored as a single vector called `LMISYS`.

There are two ways of generating the internal description of a given LMI system: (1) by a sequence of `lmivar`/`lmiterm` commands that build it incrementally, or (2) via the LMI Editor `lmiedit` where LMIs can be specified directly as symbolic matrix expressions. Though somewhat less flexible and powerful than the command-based description, the LMI Editor is more straightforward to use, hence particularly well-suited for beginners. Thanks to its coding and decoding capabilities, it also constitutes a good tutorial introduction to `lmivar` and `lmiterm`. Accordingly, beginners may elect to skip the subsections on `lmivar` and `lmiterm` and to concentrate on the GUI-based specification of LMIs with `lmiedit`.

See Also

Related Examples

- “Specify LMI System at the Command Line” on page 5-7
- “Specify LMIs with the LMI Editor GUI” on page 5-13

Specify LMI System at the Command Line

This tutorial example shows how to specify LMI systems at the command line using the LMI Lab tools.

Specify LMI System

Consider a stable transfer function,

$$G(s) = C(sI - A)^{-1}B\,.$$

Suppose that G has four inputs, four outputs, and six states. Consider also a set of input/output scaling matrices D with block-diagonal structure given by:

$$D = \begin{pmatrix} d_1 & 0 & 0 & 0 \\ 0 & d_1 & 0 & 0 \\ 0 & 0 & d_2 & d_3 \\ 0 & 0 & d_4 & d_5 \end{pmatrix}.$$

The following problem arises in the robust stability analysis of systems with time-varying uncertainty [4]. Find, if any, a scaling D with the specified structure, such that the largest gain across frequency of $DG(s)D^{-1}$ is less than 1.

This problem has a simple LMI formulation: There exists an adequate scaling D if the following feasibility problem has solutions. Find two symmetric matrices $X \in R_{6\times 6}$ and $S = D^TD \in R_{4\times 4}$ such that:

$$\begin{pmatrix} A^TX + XA + C^TSC & XB \\ B^TX & -S \end{pmatrix} < 0,$$

$$X > 0,$$

$$S > 1\,.$$

You can use the LMI Editor to specify the LMI problem described by these expressions, as shown in “Specify LMIs with the LMI Editor GUI” on page 5-13. Alternatively, define it at the command line using `lmivar` and `lmiterm`, as follows.

For this example, use the following values for A, B, and C.

```
A = [  -0.8715    0.5202    0.7474    1.0778   -0.9686    0.1005;
       -0.5577   -1.0843    1.8912    0.2523    1.0641   -0.0345;
       -0.2615   -1.7539   -1.5452   -0.2143    0.0923   -2.4192;
        0.6087   -1.0741    0.1306   -2.5575    2.3213    0.2388;
       -0.7169    0.3582   -1.4195    1.7043   -2.6530   -1.4276;
       -1.2944   -0.6752    1.6983    1.6764   -0.3646   -1.7730 ];

B = [        0    0.8998   -0.2130    0.9835;
             0   -0.3001         0   -0.2977;
      -1.0322         0   -1.0431    1.1437;
             0   -0.3451   -0.2701   -0.5316;
      -0.4189    1.0128   -0.4381         0;
            0         0   -0.4087         0];

C = [      0    2.0034         0    1.0289    0.1554    0.7135;
      0.9707    0.9510    0.7059    1.4580   -1.2371    0.3174;
           0         0    1.4158    0.0475   -2.1935    0.4136;
     -0.4383    0.6489   -1.6045    1.7463   -0.3334   -0.5771];
```

Define the LMI variables X and S, and then specify the terms of each LMI.

```
setlmis([])
X = lmivar(1,[6 1]);
S = lmivar(1,[2 0;2 1]);

% 1st LMI
lmiterm([1 1 1 X],1,A,'s');
lmiterm([1 1 1 S],C',C);
lmiterm([1 1 2 X],1,B);
lmiterm([1 2 2 S],-1,1);

% 2nd LMI
lmiterm([-2 1 1 X],1,1);

% 3rd LMI
lmiterm([-3 1 1 S],1,1);
lmiterm([3 1 1 0],1);

LMISYS = getlmis;
```

The `lmivar` commands define the two matrix variables, *X* and *S*. The `lmiterm` commands describe the terms in each LMI. `getlmis` returns the internal representation `LMISYS` of this LMI problem.

For more details on how to use these commands, see:

- “Initializing the LMI System” on page 5-8
- “Specifying the LMI Variables” on page 5-9
- “Specifying Individual LMIs” on page 5-10

For more information about how `lmivar` updates the internal representation of the LMI problem, see “How lmivar and lmiterm Manage LMI Representation” on page 5-16.

Initializing the LMI System

The description of an LMI system should begin with `setlmis` and end with `getlmis`. The function `setlmis` initializes the LMI system description. When specifying a new system, type

```
setlmis([])
```

To add on to an existing LMI system with internal representation `LMIS0`, type

```
setlmis(LMIS0)
```

Specifying the LMI Variables

The matrix variables are declared one at a time with `lmivar` and are characterized by their structure. To facilitate the specification of this structure, the LMI Lab offers two predefined structure types along with the means to describe more general structures:

Type 1	Symmetric block diagonal structure. This corresponds to matrix variables of the form $$X = \begin{pmatrix} D_1 & 0 & \dots & 0 \\ 0 & D_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & D_r \end{pmatrix}$$ where each diagonal block D_j is square and is either zero, a *full symmetric* matrix, or a *scalar* matrix $D_j = d \times I, \quad d \epsilon R$ This type encompasses ordinary symmetric matrices (single block) and scalar variables (one block of size one).
Type 2	**Rectangular** structure. This corresponds to arbitrary rectangular matrices without any particular structure.
Type 3	**General** structures. This third type is used to describe more sophisticated structures and/or correlations between the matrix variables. The principle is as follows: each entry of X is specified independently as either 0, xn, or –xn where xn denotes the n-th decision variable in the problem. For details on how to use Type 3, see "Structured Matrix Variables" on page 5-27 as well as the `lmivar` entry in the reference pages.

In "Specify LMI System" on page 5-7, the matrix variables X and S are of Type 1. Indeed, both are symmetric and S inherits the block-diagonal structure of D. Specifically, S is of the form

$$S = \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_2 & s_3 \\ 0 & 0 & s_3 & s_4 \end{pmatrix}.$$

Initialize the description and declare these two matrix variables.

```
setlmis([])
lmivar(1,[6 1]);     % X
lmivar(1,[2 0;2 1]); % S
```

In both `lmivar` commands, the first input specifies the structure type and the second input contains additional information about the structure of the variable:

- For a matrix variable X of Type 1, this second input is a matrix with two columns and as many rows as diagonal blocks in X. The first column lists the sizes of the diagonal blocks and the second column specifies their nature with the following convention:

  1: full symmetric block

0: scalar block

–1: zero block

In the second command, for instance, `[2 0;2 1]` means that *S* has two diagonal blocks, the first one being a 2-by-2 scalar block and the second one a 2-by-2 full block.

- For matrix variables of Type 2, the second input of `lmivar` is a two-entry vector listing the row and column dimensions of the variable. For instance, a 3-by-5 rectangular matrix variable would be defined by

```
lmivar(2,[3 5])
```

For convenience, `lmivar` also returns a “tag” that identifies the matrix variable for subsequent reference. For instance, *X* and *S* in “Specify LMI System” on page 5-7 could be defined by

```
X = lmivar(1,[6 1]);
S = lmivar(1,[2 0;2 1]);
```

The identifiers *X* and *S* are integers corresponding to the ranking of *X* and *S* in the list of matrix variables (in the order of declaration). Here their values would be `X=1` and `S=2`. Note that these identifiers still point to *X* and *S* after deletion or instantiation of some of the matrix variables. Finally, `lmivar` can also return the total number of decision variables allocated so far as well as the entry-wise dependence of the matrix variable on these decision variables (see the `lmivar` entry in the reference pages for more details).

Specifying Individual LMIs

After declaring the matrix variables with `lmivar`, we are left with specifying the term content of each LMI. Recall that LMI terms fall into three categories:

- The *constant terms*, i.e., fixed matrices like *I* in the left side of the LMI $S > I$.
- The *variable terms*, i.e., terms involving a matrix variable. For instance, A^TX and C^TSC in the expression:

$$\begin{pmatrix} A^TX + XA + C^TSC & XB \\ B^TX & -S \end{pmatrix} < 0$$

  Variable terms are of the form *PXQ* where *X* is a variable and *P, Q* are given matrices called the left and right *coefficients*, respectively.
- The *outer factors*.

When describing the term content of an LMI, specify only the terms in the blocks on or above the diagonal. The inner factors being symmetric, this is sufficient to specify the entire LMI. *Specifying all blocks results in the duplication of off-diagonal terms, hence in the creation of a different LMI.* Alternatively, you can describe the blocks on or below the diagonal.

LMI terms are specified one at a time with `lmiterm`. For instance, the LMI

$$\begin{pmatrix} A^TX + XA + C^TSC & XB \\ B^TX & -S \end{pmatrix} < 0$$

is described by

```
lmiterm([1 1 1 1],1,A,'s');
lmiterm([1 1 1 2],C',C);
lmiterm([1 1 2 1],1,B);
lmiterm([1 2 2 2],-1,1);
```

These commands successively declare the terms $A^TX + XA$, C^TSC, XB, and $-S$. In each command, the first argument is a four-entry vector listing the term characteristics as follows:

- The first entry indicates to which LMI the term belongs. The value `m` means "left side of the *m*-th LMI," and `−m` means "right side of the *m*-th LMI."
- The second and third entries identify the block to which the term belongs. For instance, the vector `[1 1 2 1]` indicates that the term is attached to the (1, 2) block.
- The last entry indicates which matrix variable is involved in the term. This entry is `0` for constant terms, `k` for terms involving the *k*-th matrix variable X_k, and `−k` for terms involving X_k^T (here X and S are first and second variables in the order of declaration).

Finally, the second and third arguments of `lmiterm` contain the numerical data (values of the constant term, outer factor, or matrix coefficients P and Q for variable terms PXQ or PX^TQ). These arguments must refer to existing MATLAB variables and be *real-valued*. See "Complex-Valued LMIs" on page 5-29 for the specification of LMIs with complex-valued coefficients.

Some shorthand is provided to simplify term specification. First, blocks are zero by default. Second, in *diagonal blocks* the extra argument `'s'` allows you to specify the conjugated expression $AXB + B^TX^TA^T$ with a *single* `lmiterm` command. For instance, the first command specifies $A^TX + XA$ as the "symmetrization" of XA. Finally, scalar values are allowed as shorthand for *scalar matrices*, i.e., matrices of the form αI with α scalar. Thus, a constant term of the form αI can be specified as the "scalar" α. This also applies to the coefficients P and Q of variable terms. The dimensions of scalar matrices are inferred from the context and set to 1 by default. For instance, the third LMI $S > I$ in "Specify Matrix Variable Structures" on page 5-27 is described by

```
lmiterm([-3 1 1 2],1,1);      % 1*S*1 = S
lmiterm([3 1 1 0],1);         % 1*I = I
```

Recall that by convention `S` is considered as the right side of the inequality, which justifies the –3 in the first command.

Finally, to improve readability it is often convenient to attach an identifier (tag) to each LMI and matrix variable. The variable identifiers are returned by `lmivar` and the LMI identifiers are set by the function `newlmi`. These identifiers can be used in `lmiterm` commands to refer to a given LMI or matrix variable. For the LMI system of "Specify LMI System" on page 5-7, this would look like:

```
setlmis([])
X = lmivar(1,[6 1]);
S = lmivar(1,[2 0;2 1]);

BRL = newlmi;
lmiterm([BRL 1 1 X],1,A,'s');
lmiterm([BRL 1 1 S],C',C);
lmiterm([BRL 1 2 X],1,B);
lmiterm([BRL 2 2 S],-1,1);

Xpos = newlmi;
lmiterm([-Xpos 1 1 X],1,1);

Slmi = newlmi;
```

```
lmiterm([-Slmi 1 1 S],1,1);
lmiterm([Slmi 1 1 0],1);
```

When the LMI system is completely specified, get the internal representation of the problem.

```
LMISYS = getlmis;
```

This returns the internal representation `LMISYS` of this LMI system. This MATLAB description of the problem can be forwarded to other LMI-Lab functions for subsequent processing. The command `getlmis` must be used *only once* and after declaring *all* matrix variables and LMI terms.

Here the identifiers `X` and `S` point to the variables *X* and *S* while the tags `BRL`, `Xpos`, and `Slmi` point to the first, second, and third LMI, respectively. Note that –`Xpos` refers to the right-hand side of the second LMI. Similarly, –`X` would indicate transposition of the variable *X*.

See Also

`getlmis` | `lmiterm` | `lmivar` | `setlmis`

Related Examples

- “Specify LMIs with the LMI Editor GUI” on page 5-13

More About

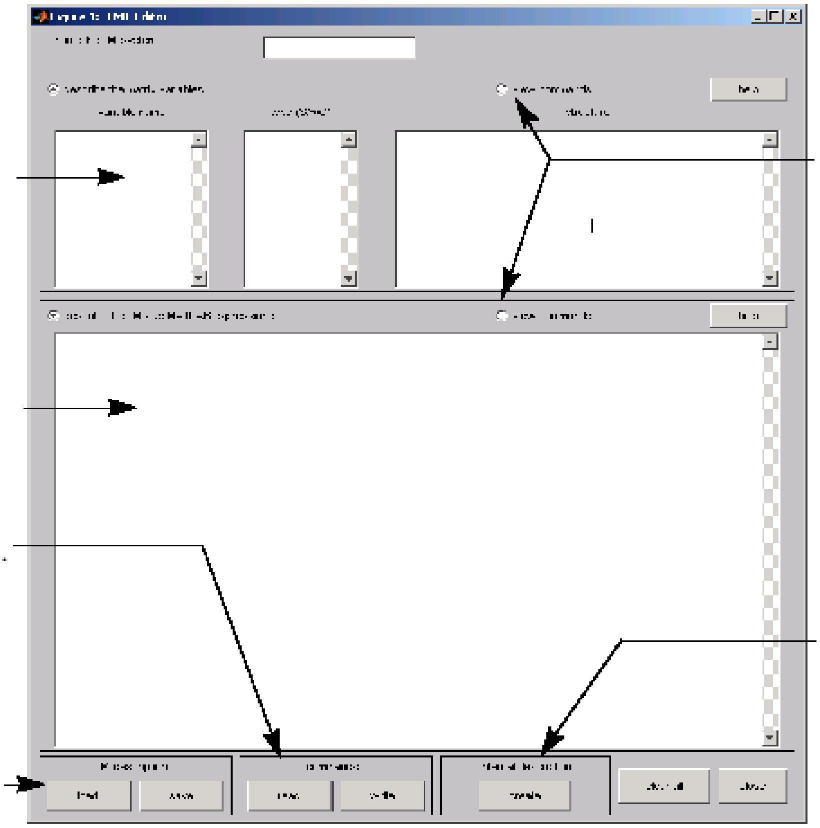
- “How lmivar and lmiterm Manage LMI Representation” on page 5-16

Specify LMIs with the LMI Editor GUI

The LMI Editor `lmiedit` is a graphical user interface (GUI) to specify LMI systems in a straightforward symbolic manner. Typing

```
lmiedit
```

calls up a window with several editable text areas and various buttons.



Declare each matrix variable (name and structure) here.

Specify the LMIs as MATLAB expressions here.

Read a sequence of lmivar/lmiterm commands from a file by clicking **Read**.

Save/Load the symbolic description of the LMI system as a MATLAB string.

Use the **view commands** buttons to visualize the sequence of lmivar/lmiterm commands needed to describe this LMI system.

Generate the internal representation of the LMI system by clicking **Create**.

In more detail, to specify your LMI system,

1 Declare each matrix variable (name and structure) in the upper half of the worksheet. 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. This matrix contains specific information about the structure and corresponds to the second argument of `lmivar` (see "Specifying the LMI Variables" on page 5-9 for details).

Please use *one line per matrix variable* in the text editing areas.

2 Specify the LMIs as MATLAB expressions in the lower half of the worksheet. For instance, the LMI

$$\begin{pmatrix} A^TX + XA & XB \\ B^TX & -I \end{pmatrix} < 0$$

is entered by typing

```
[a'*x+x*a x*b; b'*x -1] < 0
```

if `x` is the name given to the matrix variable *X* in the upper half of the worksheet. The left- and right-hand sides of the LMIs should be *valid* MATLAB expressions.

Once the LMI system is fully specified, the following tasks can be performed by clicking the corresponding button:

- Visualize the sequence of `lmivar`/`lmiterm` commands needed to describe this LMI system (`view commands` button). Conversely, the LMI system defined by a particular sequence of `lmivar`/`lmiterm` commands can be displayed as a MATLAB expression by clicking on the **describe...** buttons.

  Beginners can use this facility as a tutorial introduction to the `lmivar` and `lmiterm` commands.
- Save the symbolic description of the LMI system (`save` button). This description can be reloaded later on by clicking the **load** button.
- Read a sequence of `lmivar`/`lmiterm` commands from a file (**read** button). You can then click on **describe the matrix variables** or **describe the LMIs** to visualize the symbolic expression of the LMI system specified by these commands. The file should describe a single LMI system but may otherwise contain any sequence of MATLAB commands.

  This feature is useful for code validation and debugging.

  Write in a file the sequence of `lmivar`/`lmiterm` commands needed to describe a particular LMI system (**write** button).

  This is helpful to develop code and prototype MATLAB functions based on the LMI Lab.
- Generate the internal representation of the LMI system by clicking **create**. The result is written in a MATLAB variable named after the LMI system (if the “name of the LMI system” is set to `mylmi`, the internal representation is written in the MATLAB variable `mylmi`). Note that all LMI-related data should be defined in the MATLAB workspace at this stage.

  The internal representation can be passed directly to the LMI solvers or any other LMI Lab function.

Keyboard Shortcuts

As with `lmiterm`, you can use various shortcuts when entering LMI expressions at the keyboard. For instance, zero blocks can be entered simply as `0` and need not be dimensioned. Similarly, the identity matrix can be entered as `1` without dimensioning. Finally, *upper diagonal* LMI blocks need not be fully specified. Rather, you can just type (*) in place of each such block.

Limitations

Though fairly general, `lmiedit` is not as flexible as `lmiterm` and the following limitations should be kept in mind:

- Parentheses cannot be used around matrix variables. For instance, the expression

  ```
  (a*x+b)'*c + c'*(a*x+b)
  ```

  is invalid when *x* is a variable name. By contrast,

```
(a+b)'*x + x'*(a+b)
```

is perfectly valid.

- Loops and `if` statements are ignored.
- When turning `lmiterm` commands into a symbolic description of the LMI system, an error is issued if the first argument of `lmiterm` cannot be evaluated. Use the LMI and variable identifiers supplied by `newlmi` and `lmivar` to avoid such difficulties.

See Also

Related Examples

- "Specify LMI System at the Command Line" on page 5-7

How lmivar and lmiterm Manage LMI Representation

Users familiar with MATLAB may wonder how `lmivar` and `lmiterm` physically update the internal representation `LMISYS` since `LMISYS` is not an argument to these functions. In fact, all updating is performed through global variables for maximum speed. These global variables are initialized by `setlmis`, cleared by `getlmis`, and are not visible in the workspace. Even though this artifact is transparent from the user's viewpoint, be sure to:

- Invoke `getlmis` only once and after completely specifying the LMI system.
- Refrain from using the command `clear global` before the LMI system description is ended with `getlmis`.

See Also

More About

- "Querying the LMI System Description" on page 5-17

Querying the LMI System Description

Recall that the full description of an LMI system is stored as a single vector called the internal representation. The user should not attempt to read or retrieve information directly from this vector. Robust Control Toolbox software provides three functions called `lmiinfo`, `lminbr`, and `matnbr` to extract and display all relevant information in a user-readable format.

lmiinfo

`lminbr` is an interactive facility to retrieve qualitative information about LMI systems. This includes the number of LMIs, the number of matrix variables and their structure, the term content of each LMI block, etc. To invoke `lmiinfo`, enter

```
lmiinfo(LMISYS)
```

where `LMISYS` is the internal representation of the LMI system produced by `getlmis`.

lminbr and matnbr

These two functions return the number of LMIs and the number of matrix variables in the system. To get the number of matrix variables, for instance, enter

```
matnbr(LMISYS)
```

See Also

More About

- “LMI Solvers” on page 5-18

LMI Solvers

LMI solvers are provided for the following three generic optimization problems (here x denotes the vector of decision variables, i.e., of the free entries of the matrix variables $X_1, \ldots, X_K$):

- Feasibility problem

  Find $x \in R^N$ (or equivalently matrices $X_1, \ldots, X_K$ with prescribed structure) that satisfies the LMI system

  $A(x) < B(x)$

  The corresponding solver is called `feasp`.
- Minimization of a linear objective under LMI constraints

  Minimize $c^T x$ over $x \in R^N$ subject to $A(x) < B(x)$

  The corresponding solver is called `mincx`.
- Generalized eigenvalue minimization problem

  Minimize λ over $x \in R^N$ subject to

  $$\begin{aligned} C(x) &< D(x) \\ 0 &< B(x) \\ A(x) &< \lambda B(x). \end{aligned}$$

  The corresponding solver is called `gevp`.

Note that $A(x) < \mathrm{B}(x)$ above is a shorthand notation for general structured LMI systems with decision variables $x = (x_1, \ldots, x_N)$.

The three LMI solvers `feasp`, `mincx`, and `gevp` take as input the internal representation `LMISYS` of an LMI system and return a feasible or optimizing value x^* of the decision variables. The corresponding values of the matrix variables $X_1, \ldots, X_K$ are derived from x^* with the function `dec2mat`. These solvers are C-MEX implementations of the polynomial-time Projective Algorithm Projective Algorithm of Nesterov and Nemirovski [3], [2].

For generalized eigenvalue minimization problems, it is necessary to distinguish between the standard LMI constraints $\mathrm{C}(x) < \mathrm{D}(x)$ and the linear-fractional LMIs

$\mathrm{A(x)} < \lambda \mathrm{B(x)}$

attached to the minimization of the generalized eigenvalue λ. When using `gevp`, you should follow these three rules to ensure proper specification of the problem:

- Specify the LMIs involving λ as $A(x) < B(x)$ (*without* the λ)
- Specify them *last* in the LMI system. `gevp` systematically assumes that the last L LMIs are linear-fractional if L is the number of LMIs involving λ
- Add the constraint $0 < B(x)$ or any other constraint that enforces it. This positivity constraint is required for well-posedness of the problem and is not automatically added by `gevp`.

An initial guess `xinit` for x can be supplied to `mincx` or `gevp`. Use `mat2dec` to derive `xinit` from given values of the matrix variables $X_1, \ldots, X_K$.

The example "Minimize Linear Objectives under LMI Constraints" on page 5-20 illustrates the use of the `mincx` solver.

See Also

Related Examples

- "Minimize Linear Objectives under LMI Constraints" on page 5-20

Minimize Linear Objectives under LMI Constraints

Consider the optimization problem:

Minimize Trace(X) subject to

$$A^TX + XA + XBB^TX + Q < 0 \tag{5-4}$$

with data

$$A = \begin{pmatrix} -1 & -2 & 1 \\ 3 & 2 & 1 \\ 1 & -2 & -1 \end{pmatrix}; \quad B = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}; \quad Q = \begin{pmatrix} 1 & -1 & 0 \\ -1 & -3 & -12 \\ 0 & -12 & -36 \end{pmatrix}.$$

It can be shown that the minimizer X^* is simply the stabilizing solution of the algebraic Riccati equation

$$A^TX + XA + XBB^TX + Q = 0$$

This solution can be computed directly with the Riccati solver `care` and compared to the minimizer returned by `mincx`.

From an LMI optimization standpoint, the problem specified in “Equation 5-4” is equivalent to the following linear objective minimization problem:

Minimize Trace(X) subject to

$$\begin{pmatrix} A^TX + XA + Q & XB \\ B^TX & -I \end{pmatrix} < 0. \tag{5-5}$$

Since Trace(X) is a linear function of the entries of X, this problem falls within the scope of the `mincx` solver and can be numerically solved as follows:

1. Define the LMI constraint of “Equation 5-4” by the sequence of commands

   ```
   setlmis([])
   X = lmivar(1,[3 1]) % variable X, full symmetric

   lmiterm([1 1 1 X],1,a,'s')
   lmiterm([1 1 1 0],q)
   lmiterm([1 2 2 0],-1)
   lmiterm([1 2 1 X],b',1)

   LMIs = getlmis
   ```
2. Write the objective Trace(X) as c^Tx where x is the vector of free entries of X. Since c should select the diagonal entries of X, it is obtained as the decision vector corresponding to $X = I$, that is,

   ```
   c = mat2dec(LMIs,eye(3))
   ```

   Note that the function `defcx` provides a more systematic way of specifying such objectives (see “Specifying cTx Objectives for mincx” on page 5-30 for details).
3. Call `mincx` to compute the minimizer `xopt` and the global minimum `copt = c'*xopt` of the objective:

```
options = [1e-5,0,0,0,0]
[copt,xopt] = mincx(LMIs,c,options)
```

Here `1e–5` specifies the desired relative accuracy on `copt`.

The following trace of the iterative optimization performed by `mincx` appears on the screen:

```
Solver for linear objective minimization under LMI constraints
Iterations     :     Best objective value so far
```

1		
2	-8.511476	
3	-13.063640	
***	new lower bound:	-34.023978
4	-15.768450	
***	new lower bound:	-25.005604
5	-17.123012	
***	new lower bound:	-21.306781
6	-17.882558	
***	new lower bound:	-19.819471
7	-18.339853	
***	new lower bound:	-19.189417
8	-18.552558	
***	new lower bound:	-18.919668
9	-18.646811	
***	new lower bound:	-18.803708
10	-18.687324	
***	new lower bound:	-18.753903
11	-18.705715	
***	new lower bound:	-18.732574
12	-18.712175	
***	new lower bound:	-18.723491
13	-18.714880	
***	new lower bound:	-18.719624
14	-18.716094	
***	new lower bound:	-18.717986
15	-18.716509	
***	new lower bound:	-18.717297
16	-18.716695	
***	new lower bound:	-18.716873

```
Result: feasible solution of required accuracy
    best objective value:     -18.716695
    guaranteed relative accuracy:     9.50e-06
    f-radius saturation: 0.000% of R = 1.00e+09
```

The iteration number and the best value of $c^T x$ at the current iteration appear in the left and right columns, respectively. Note that no value is displayed at the first iteration, which means that a feasible x satisfying the constraint ("Equation 5-5") was found only at the second iteration. Lower bounds on the global minimum of $c^T x$ are sometimes detected as the optimization progresses. These lower bounds are reported by the message

```
*** new lower bound: xxx
```

Upon termination, `mincx` reports that the global minimum for the objective Trace(X) is –18.716695 with relative accuracy of at least 9.5×10^{-6}. This is the value `copt` returned by `mincx`.

4 `mincx` also returns the optimizing vector of decision variables `xopt`. The corresponding optimal value of the matrix variable X is given by

```
Xopt = dec2mat(LMIs,xopt,X)
```

which returns

$$X_{opt} = \begin{pmatrix} -6.3542 & -5.8895 & 2.2046 \\ -5.8895 & -6.2855 & 2.2201 \\ 2.2046 & 2.2201 & -6.0771 \end{pmatrix}.$$

This result can be compared with the stabilizing Riccati solution computed by `care`:

```
Xst = care(a,b,q,-1)
norm(Xopt-Xst)

ans =
    6.5390e-05
```

See Also

Related Examples

- "Conversion Between Decision and Matrix Variables" on page 5-23

Conversion Between Decision and Matrix Variables

While LMIs are specified in terms of their matrix variables $X_1, . . . , X_K$, the LMI solvers optimize the vector x of free scalar entries of these matrices, called the decision variables. The two functions `mat2dec` and `dec2mat` perform the conversion between these two descriptions of the problem variables.

Consider an LMI system with three matrix variables X_1, X_2, X_3. Given particular values `X1`, `X2`, `X3` of these variables, the corresponding value `xdec` of the vector of decision variables is returned by `mat2dec`:

```
xdec = mat2dec(LMISYS,X1,X2,X3)
```

An error is issued if the number of arguments following `LMISYS` differs from the number of matrix variables in the problem (see `matnbr`).

Conversely, given a value `xdec` of the vector of decision variables, the corresponding value of the k-th matrix is given by `dec2mat`. For instance, the value `X2` of the second matrix variable is extracted from `xdec` by

```
X2 = dec2mat(LMISYS,xdec,2)
```

The last argument indicates that the second matrix variable is requested. It could be set to the matrix variable identifier returned by `lmivar`.

The total numbers of matrix variables and decision variables are returned by `matnbr` and `decnbr`, respectively. In addition, the function `decinfo` provides precise information about the mapping between decision variables and matrix variable entries.

See Also

Related Examples

- “Validating Results” on page 5-24

Validating Results

The LMI Lab offers two functions to analyze and validate the results of an LMI optimization. The function `evallmi` evaluates all variable terms in an LMI system for a given value of the vector of decision variables, for instance, the feasible or optimal vector returned by the LMI solvers. Once this evaluation is performed, the left and right sides of a particular LMI are returned by `showlmi`.

In the LMI problem considered in “Minimize Linear Objectives under LMI Constraints” on page 5-20, you can verify that the minimizer `xopt` returned by `mincx` satisfies the LMI constraint (“Equation 5-5”) as follows:

```
evlmi = evallmi(LMIs,xopt)
[lhs,rhs] = showlmi(evlmi,1)
```

The first command evaluates the system for the value `xopt` of the decision variables, and the second command returns the left and right sides of the first (and only) LMI. The negative definiteness of this LMI is checked by

```
eig(lhs-rhs)

ans =
    -2.0387e-04
    -3.9333e-05
    -1.8917e-07
    -4.6680e+01
```

See Also

Related Examples

- “Modify a System of LMIs” on page 5-25

Modify a System of LMIs

Once specified, a system of LMIs can be modified in several ways with the functions `dellmi`, `delmvar`, and `setmvar`.

Deleting an LMI

The first possibility is to remove an entire LMI from the system with `dellmi`. For instance, suppose that the LMI system of “Specify LMI System” on page 5-7 is described in `LMISYS` and that we want to remove the positivity constraint on X. This is done by

```
NEWSYS = dellmi(LMISYS,2)
```

where the second argument specifies deletion of the second LMI. The resulting system of two LMIs is returned in `NEWSYS`.

The LMI identifiers (*initial* ranking of the LMI in the LMI system) are not altered by deletions. As a result, the last LMI

$S > I$

remains known as the third LMI even though it now ranks second in the modified system. To avoid confusion, it is safer to refer to LMIs via the identifiers returned by `newlmi`. If `BRL`, `Xpos`, and `Slmi` are the identifiers attached to the three LMIs described in “Specify LMI System” on page 5-7, `Slmi` keeps pointing to $S > I$ even after deleting the second LMI by

```
NEWSYS = dellmi(LMISYS,Xpos)
```

Deleting a Matrix Variable

Another way of modifying an LMI system is to delete a matrix variable, that is, to remove all variable terms involving this matrix variable. This operation is performed by `delmvar`. For instance, consider the LMI

$A^TX + XA + BW + W^TB^T + I < 0$

with variables $X = X^T \in R^4\times^4$ and $W \in R^2\times^4$. This LMI is defined by

```
setlmis([])
X = lmivar(1,[4 1])     % X
W = lmivar(2,[2 4])     % W

lmiterm([1 1 1 X],1,A,'s')
lmiterm([1 1 1 W],B,1,'s')
lmiterm([1 1 1 0],1)

LMISYS = getlmis
```

To delete the variable `W`, type the command

```
NEWSYS = delmvar(LMISYS,W)
```

The resulting `NEWSYS` now describes the Lyapunov inequality

$A^TX + XA + I < 0$

Note that `delmvar` automatically removes all LMIs that depended only on the deleted matrix variable.

The matrix variable identifiers are not affected by deletions and continue to point to the same matrix variable. For subsequent manipulations, it is therefore advisable to refer to the remaining variables through their identifier. Finally, note that deleting a matrix variable is equivalent to setting it to the zero matrix of the same dimensions with `setmvar`.

Instantiating a Matrix Variable

The function `setmvar` is used to set a matrix variable to some given value. As a result, this variable is removed from the problem and all terms involving it become constant terms. This is useful, for instance, to fix`setmvar` some variables and optimize with respect to the remaining ones.

Consider again “Specify LMI System” on page 5-7 and suppose we want to know if the peak gain of *G* itself is less than one, that is, if

$\|G\|^{\infty} < 1$

This amounts to setting the scaling matrix *D* (or equivalently, $S = D^{T}D$) to a multiple of the identity matrix. Keeping in mind the constraint $S > I$, a legitimate choice is *S* = 2-βψ-*I*. To set *S* to this value, enter

```
NEWSYS = setmvar(LMISYS,S,2)
```

The second argument is the variable identifier `S`, and the third argument is the value to which *S* should be set. Here the value 2 is shorthand for 2-by-*I*. The resulting system `NEWSYS` reads

$$\begin{pmatrix} A^TX + XA + 2C^{TC} & XB \\ B^TX & -2I \end{pmatrix} < 0$$

$$X > 0$$

$$2I > I\,.$$

Note that the last LMI is now free of variable and trivially satisfied. It could, therefore, be deleted by

```
NEWSYS = dellmi(NEWSYS,3)
```

or

```
NEWSYS = dellmi(NEWSYS,Slmi)
```

if `Slmi` is the identifier returned by `newlmi`.

See Also

Related Examples

- “Advanced LMI Techniques” on page 5-27

Advanced LMI Techniques

This last section gives a few hints for making the most out of the LMI Lab. It is directed toward users who are comfortable with the basics, as described in “Tools for Specifying and Solving LMIs” on page 5-2.

Structured Matrix Variables

Fairly complex matrix variable structures and interdependencies can be specified with `lmivar`. Recall that the symmetric block-diagonal or rectangular structures are covered by Types 1 and 2 of `lmivar` provided that the matrix variables are independent. To describe more complex structures or correlations between variables, you must use Type 3 and specify each entry of the matrix variables directly in terms of the free scalar variables of the problem (the so-called decision variables).

With Type 3, each entry is specified as either 0 or $\pm x_n$ where x_n is the *n*-th decision variable. The following examples illustrate how to specify nontrivial matrix variable structures with `lmivar`. The following examples show variable structures with uncorrelated and interdependent matrix variables.

Specify Matrix Variable Structures

Suppose that the variables of the problem include a 3-by-3 symmetric matrix *X* and a 3-by-3 symmetric Toeplitz matrix, *Y*, given by:

$$Y = \begin{pmatrix} y_1 & y_2 & y_3 \\ y_2 & y_1 & y_2 \\ y_3 & y_2 & y_1 \end{pmatrix}.$$

The variable *Y* has three independent entries, and thus involves three decision variables. Since *Y* is independent of *X*, label these decision variables $n + 1$, $n + 2$, and $n + 3$, where *n* is the number of decision variables involved in *X*. To retrieve this number, define the Type 1 variable *X*.

```
setlmis([])
[X,n] = lmivar(1,[3 1]);
n
```

```
n = 6
```

The second output argument `n` gives the total number of decision variables used so far, which in this case is `n` = 6. Given this number, you can define *Y.*

```
Y = lmivar(3,n+[1 2 3;2 1 2;3 2 1]);
```

An equivalent expression to define Y uses the MATLAB(R) command `toeplitz` to generate the matrix.

```
Y = lmivar(3,toeplitz(n+[1 2 3]));
```

To confirm the variables, visualize the decision variable distributions in *X* and *Y* using `decinfo`.

```
lmis = getlmis;
decinfo(lmis,X)
```

```
ans = 3×3
```

```
     1     2     4
     2     3     5
     4     5     6
```

```
decinfo(lmis,Y)
```

```
ans = 3×3

     7     8     9
     8     7     8
     9     8     7
```

Specify Interdependent Matrix Variables

Consider three matrix variables, X, Y, and Z, with the following structure.

$$X = \begin{pmatrix} x & 0 \\ 0 & y \end{pmatrix}, \quad Y = \begin{pmatrix} z & 0 \\ 0 & t \end{pmatrix}, \quad Z = \begin{pmatrix} 0 & -x \\ -t & 0 \end{pmatrix},$$

where x, y, z, and t are independent scalar variables. To specify such a triple, first define the two independent variables, X and Y, which are both Type 1.

```
setlmis([]);
[X,n,sX] = lmivar(1,[1 0;1 0]);
[Y,n,sY] = lmivar(1,[1 0;1 0]);
```

The third output of lmivar gives the entry-wise dependence of X and Y on the decision variables $(x_1, x_2, x_3, x_4) := (x, y, z, t)$.

```
sX
```

```
sX = 2×2

     1     0
     0     2
```

```
sY
```

```
sY = 2×2

     3     0
     0     4
```

Using lmivar, you can now specify the structure of the Type 3 variable Z in terms of the decision variables $x_1 = x$ and $x_4 = t$.

```
[Z,n,sZ] = lmivar(3,[0 -sX(1,1);-sY(2,2) 0]);
```

Because `sX(1,1)` refers to x_1 and `sY(2,2)` refers to x_4, this expression defines the variable `Z` as:

$$Z = \begin{pmatrix} 0 & -x_1 \\ -x_4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -x \\ -t & 0 \end{pmatrix}.$$

Confirm this results by checking the entry-wise dependence of Z on its decision variables.

```
sZ
```

```
sZ = 2×2

     0    -1
    -4     0
```

Complex-Valued LMIs

The LMI solvers are written for real-valued matrices and cannot directly handle LMI problems involving complex-valued matrices. However, complex-valued LMIs can be turned into real-valued LMIs by observing that a complex Hermitian matrix $L(x)$ satisfies

$$L(x) < 0$$

if and only if

$$\begin{pmatrix} \mathrm{Re}(L(x)) & \mathrm{Im}(L(x)) \\ -\mathrm{Im}(L(x)) & \mathrm{Re}(L(x)) \end{pmatrix} < 0.$$

This suggests the following systematic procedure for turning complex LMIs into real ones:

- Decompose every complex matrix variable X as

  $$X = X_1 + jX_2$$

  where X_1 and X_2 are real
- Decompose every complex matrix coefficient A as

  $$A = A_1 + jA_2$$

  where A_1 and A_2 are real
- Carry out all complex matrix products. This yields affine expressions in X_1, X_2 for the real and imaginary parts of each LMI, and an equivalent real-valued LMI is readily derived from the above observation.

For LMIs without outer factor, a streamlined version of this procedure consists of replacing any occurrence of the matrix variable $X = X_1 + jX_2$ by

$$\begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix}$$

and any fixed matrix $A = A_1 + jA_2$, including real ones, by

$$\begin{pmatrix} A_1 & A_2 \\ -A_2 & A_1 \end{pmatrix}.$$

For instance, the real counterpart of the LMI system

$$M^H X M < X, \quad X = X^H > I \tag{5-6}$$

reads (given the decompositions $M = M_1 + jM_2$ and $X = X_1 + jX_2$ with Mj, Xj real):

$$\begin{pmatrix} M_1 & M_2 \\ -M_2 & M_1 \end{pmatrix}^T \begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix} \begin{pmatrix} M_1 & M_2 \\ -M_2 & M_1 \end{pmatrix} < \begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix}$$

$$\begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix} < I .$$

Note that $X = X^H$ in turn requires that $X_1 = X_1^H$ and $X_2 + X_2^T = 0$. Consequently, X_1 and X_2 should be declared as symmetric and skew- symmetric matrix variables, respectively.

Assuming, for instance, that $M \in \mathbf{C}^{5\times 5}$, the LMI system (“Equation 5-6”) would be specified as follows:

```
M1=real(M), M2=imag(M)
bigM=[M1 M2;-M2 M1]
setlmis([])

% declare bigX=[X1 X2;-X2 X1] with X1=X1' and X2+X2'=0:

[X1,n1,sX1] = lmivar(1,[5 1])
[X2,n2,sX2] = lmivar(3,skewdec(5,n1))
bigX = lmivar(3,[sX1 sX2;-sX2 sX1])

% describe the real counterpart of (1.12):

lmiterm([1 1 1 0],1)
lmiterm([-1 1 1 bigX],1,1)
lmiterm([2 1 1 bigX],bigM',bigM)
lmiterm([-2 1 1 bigX],1,1)

lmis = getlmis
```

Note the three-step declaration of the structured matrix variable `bigX`,

$$bigX = \begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix}.$$

1. Specify X_1 as a (real) symmetric matrix variable and save its structure description `sX1` as well as the number `n1` of decision variables used in X_1.
2. Specify X_2 as a skew-symmetric matrix variable using Type 3 of `lmivar` and the utility `skewdec`. The command `skewdec(5,n1)` creates a 5-by–5 skew-symmetric structure depending on the decision variables `n1 + 1`, `n1 + 2`,...
3. Define the structure of `bigX` in terms of the structures `sX1` and `sX2` of X_1 and X_2.

See “Structured Matrix Variables” on page 5-27 for more details on such structure manipulations.

Specifying cTx Objectives for mincx

The LMI solver `mincx` minimizes linear objectives of the form c^Tx where x is the vector of decision variables. In most control problems, however, such objectives are expressed in terms of the matrix variables rather than of x. Examples include Trace(X) where X is a symmetric matrix variable, or u^TXu where u is a given vector.

The function defcx facilitates the derivation of the *c* vector when the objective is an affine function of the *matrix variables*. For the sake of illustration, consider the linear objective

$$\mathrm{Trace}(X) + x_0^T P x_0$$

where *X* and *P* are two symmetric variables and x_0 is a given vector. If lmsisys is the internal representation of the LMI system and if x_0, *X, P* have been declared by

```
x0 = [1;1]
setlmis([])
X = lmivar(1,[3 0])
P = lmivar(1,[2 1])
    :
    :
lmisys = getlmis
```

the *c* vector such that $c^T x = \mathrm{Trace}(X) + x_0^T P x_0$ can be computed as follows:

```
n = decnbr(lmisys)
c = zeros(n,1)

for j=1:n,
    [Xj,Pj] = defcx(lmisys,j,X,P)
    c(j) = trace(Xj) + x0'*Pj*x0
end
```

The first command returns the number of decision variables in the problem and the second command dimensions *c* accordingly. Then the for loop performs the following operations:

1. Evaluate the matrix variables *X* and *P* when all entries of the decision vector *x* are set to zero except *xj*: = 1. This operation is performed by the function defcx. Apart from lmisys and j, the inputs of defcx are the identifiers X and P of the variables involved in the objective, and the outputs Xj and Pj are the corresponding values.
2. Evaluate the objective expression for X:= Xj and P:= Pj. This yields the *j*-th entry of c by definition.

In our example the result is

```
c =
    3
    1
    2
    1
```

Other objectives are handled similarly by editing the following generic skeleton:

```
n = decnbr( LMI system )
c = zeros(n,1)
for j=1:n,
    [ matrix values ] = defcx( LMI system,j,
matrix identifiers)
    c(j) = objective(matrix values)
end
```

Feasibility Radius

When solving LMI problems with `feasp`, `mincx`, or `gevp`, it is possible to constrain the solution x to lie in the ball

$$x^T x < R^2$$

where $R > 0$ is called the *feasibility radius*. This specifies a maximum (Euclidean norm) magnitude for x and avoids getting solutions of very large norm. This may also speed up computations and improve numerical stability. Finally, the feasibility radius bound regularizes problems with redundant variable sets. In rough terms, the set of scalar variables is redundant when an equivalent problem could be formulated with a smaller number of variables.

The feasibility radius R is set by the third entry of the options vector of the LMI solvers. Its default value is $R = 109$. Setting R to a negative value means “no rigid bound,” in which case the feasibility radius is increased during the optimization if necessary. This “flexible bound” mode may yield solutions of large norms.

Well-Posedness Issues

The LMI solvers used in the LMI Lab are based on interior-point optimization techniques. To compute feasible solutions, such techniques require that the system of LMI constraints be strictly feasible, that is, the feasible set has a nonempty interior. As a result, these solvers may encounter difficulty when the LMI constraints are feasible but not *strictly feasible*, that is, when the LMI

$$L(x) \le 0$$

has solutions while

$$L(x) < 0$$

has no solution.

For feasibility problems, this difficulty is automatically circumvented by `feasp`, which reformulates the problem:

Find x such that

$$L(x) \le 0 \qquad (5\text{-}7)$$

as:

Minimize t subject to

$$Lx < t \times I.$$

In this modified problem, the LMI constraint is always strictly feasible in x, t and the original LMI “Equation 5-7” is feasible if and only if the global minimum $t_{\min}$ of “Equation 5-7” satisfies

$$t_{\min} \le 0$$

For feasible but not strictly feasible problems, however, the computational effort is typically higher as `feasp` strives to approach the global optimum $t_{\min} = 0$ to a high accuracy.

For the LMI problems addressed by `mincx` and `gevp`, nonstrict feasibility generally causes the solvers to fail and to return an “infeasibility” diagnosis. Although there is no universal remedy for this difficulty, it is sometimes possible to eliminate underlying algebraic constraints to obtain a strictly feasible problem with fewer variables.

Another issue has to do with homogeneous feasibility problems such as

$A^TP + P\,A < 0, P > 0$

While this problem is technically well-posed, the LMI optimization is likely to produce solutions close to zero (the trivial solution of the nonstrict problem). To compute a nontrivial Lyapunov matrix and easily differentiate between feasibility and infeasibility, replace the constraint $P > 0$-by-$P > \alpha I$ with $\alpha > 0$. Note that this does not alter the problem due to its homogeneous nature.

Semi-Definite B(x) in gevp Problems

Consider the generalized eigenvalue minimization problem

Minimize λ subject to

$$A(x) < \lambda B(x),\ B(x) > 0,\ C(x) < 0. \qquad (5\text{-}8)$$

Technically, the positivity of $B(x)$ for some $x \in R^n$ is required for the well-posedness of the problem and the applicability of polynomial-time interior-point methods. Hence problems where

$$B(x) = \begin{pmatrix} B_1(x) & 0 \\ 0 & 0 \end{pmatrix}$$

with $B_1(x) > 0$ strictly feasible, cannot be directly solved with `gevp`. A simple remedy consists of replacing the constraints

$A(x) < B(x), B(x) > 0$

by

$$A(x) < \begin{pmatrix} Y & 0 \\ 0 & 0 \end{pmatrix}, \quad Y < \lambda B_1(x), \quad B_1(x) > 0$$

where Y is an additional symmetric variable of proper dimensions. The resulting problem is equivalent to “Equation 5-8” and can be solved directly with `gevp`.

Efficiency and Complexity Issues

As explained in “Tools for Specifying and Solving LMIs” on page 5-2, the term-oriented description of LMIs used in the LMI Lab typically leads to higher efficiency than the canonical representation

$$A_0 + x_1A_1 + \dots + x_NA_N < 0. \qquad (5\text{-}9)$$

This is no longer true, however, when the number of variable terms is nearly equal to or greater than the number N of decision variables in the problem. If your LMI problem has few free scalar variables but many terms in each LMI, it is therefore preferable to rewrite it as “Equation 5-9” and to specify it in this form. Each scalar variable x_j is then declared independently and the LMI terms are of the form x_jA_j.

If M denotes the total row size of the LMI system and N the total number of scalar decision variables, the flop count per iteration for the `feasp` and `mincx` solvers is proportional to

- N^3 when the least-squares problem is solved via Cholesky factorization of the Hessian matrix (default) [2].
- M-by-N^2 when numerical instabilities warrant the use of QR factorization instead.

While the theory guarantees a worst-case iteration count proportional to M, the number of iterations actually performed grows slowly with M in most problems. Finally, while `feasp` and `mincx` are comparable in complexity, `gevp` typically demands more computational effort. Make sure that your LMI problem cannot be solved with `mincx` before using `gevp`.

Solving M + PTXQ + QTXTP < 0

In many output-feedback synthesis problems, the design can be performed in two steps:

1. Compute a closed-loop Lyapunov function via LMI optimization.
2. Given this Lyapunov function, derive the controller state-space matrices by solving an LMI of the form

   $$M + P^T X Q + Q^T X^T P < 0 \qquad (5\text{-}10)$$

   where M, P, Q are given matrices and X is an unstructured m-by-n matrix variable.

It turns out that a particular solution Xc of “Equation 5-10” can be computed via simple linear algebra manipulations [1]. Typically, Xc corresponds to the center of the ellipsoid of matrices defined by “Equation 5-10”.

The function `basiclmi` returns the “explicit” solution Xc:

```
Xc = basiclmi(M,P,Q)
```

Since this central solution sometimes has large norm, `basiclmi` also offers the option of computing an approximate least-norm solution of “Equation 5-10”. This is done by

```
X = basiclmi(M,P,Q,'Xmin')
```

and involves LMI optimization to minimize $\|X\|$.

See Also

More About

- “Tools for Specifying and Solving LMIs” on page 5-2

Bibliography

[1] Gahinet, P., and P. Apkarian, “A Linear Matrix Inequality Approach to H^∞ Control,” *Int. J. Robust and Nonlinear Contr.*, 4 (1994), pp. 421-448.

[2] Nemirovski, A., and P. Gahinet, “The Projective Method for Solving Linear Matrix Inequalities,” *Proc. Amer. Contr. Conf.*, 1994, pp. 840-844.

[3] Nesterov, Yu, and A. Nemirovski, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, SIAM Books, Philadelphia, 1994.

[4] Shamma, J.S., “Robustness Analysis for Time-Varying Systems,” *Proc. Conf. Dec. Contr.*, 1992, pp. 3163-3168.

Analyzing Uncertainty Effects in Simulink

Analyzing Uncertainty in Simulink

Robust Control Toolbox software provides tools to model uncertain dynamics in Simulink. Using these tools, you can analyze how the uncertainty impacts the time-domain and frequency-domain behavior of a Simulink model.

The Uncertain State Space block, included in the Robust Control Toolbox block library, is a convenient way to incorporate uncertainty information in a Simulink model. For more information, see “Specify Uncertainty Using Uncertain State Space Blocks” on page 6-4. Using this block, you can perform the following types of analysis:

- Vary the uncertainty and see how it affects the time responses (Monte Carlo analysis). See “Simulate Uncertainty Effects” on page 6-7.
- Analyze the effects of uncertainty on the linearized dynamics:
  - If the operating point does not depend on the parameter uncertainty, use `ulinearize` to obtain an uncertain state-space model. You can then use `usample` to sample the uncertain variables and obtain a family of LTI models.
  - If the operating point depends on the parameter uncertainty, use `usample` to sample the uncertainty and then use the Simulink Control Design™ `linearize` command to compute the linearized dynamics for each uncertainty value.

    See “How to Vary Uncertainty Values” on page 6-7 and “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19.
- Compute an *uncertain linearization*, i.e., obtain an uncertain state-space model (`uss` object) that combines the uncertain variables with the linearized dynamics. You can use this model to perform worst-case robustness analysis. See “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19.

If you cannot use Uncertain State Space blocks in the Simulink model because you share the model or generate code, you can still compute an uncertain linearization by specifying a block to linearize to an uncertain variable. For example, you can specify a gain block to linearize to an uncertain real parameter (`ureal`). See “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 6-19. You can then use the uncertain state-space model to analyze robustness in the linear operating range.

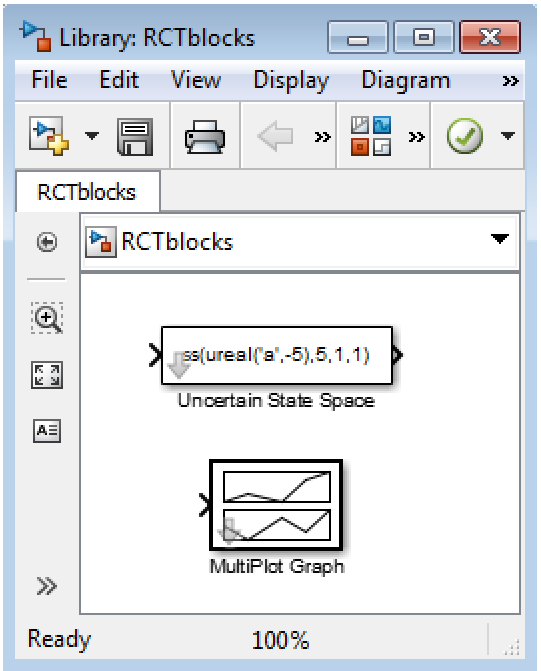
Simulink Blocks for Analyzing Uncertainty

Robust Control Toolbox software provides an Uncertain State Space block to model parametric and dynamic uncertainty in Simulink. The block library also contains a MultiPlot Graph block that you use with the Uncertain State Space block to plot and visualize Monte Carlo simulation responses.

To open the Robust Control Toolbox block library, type the following command at the MATLAB prompt:

```
RCTblocks
```

The block library opens, as shown in the following figure.



Alternatively, in a Simulink model window, click ▦ to launch to Library Browser. In the Library Browser, select **Robust Control Toolbox**.

See Also

Uncertain State Space | `linearize` | `ulinearize`

Related Examples

- “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19

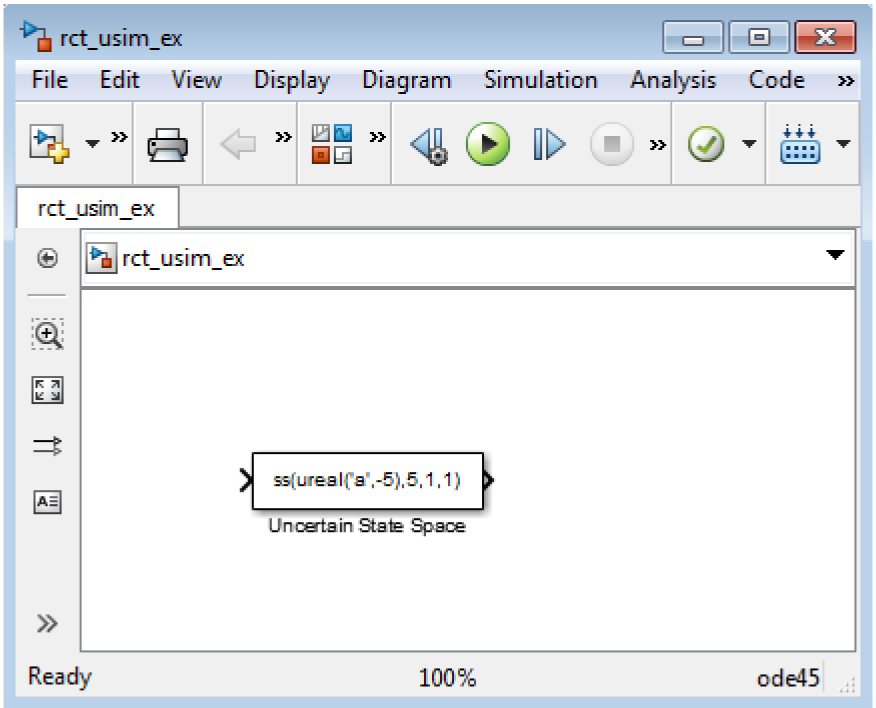
Specify Uncertainty Using Uncertain State Space Blocks

How to Specify Uncertainty in Uncertain State Space Blocks

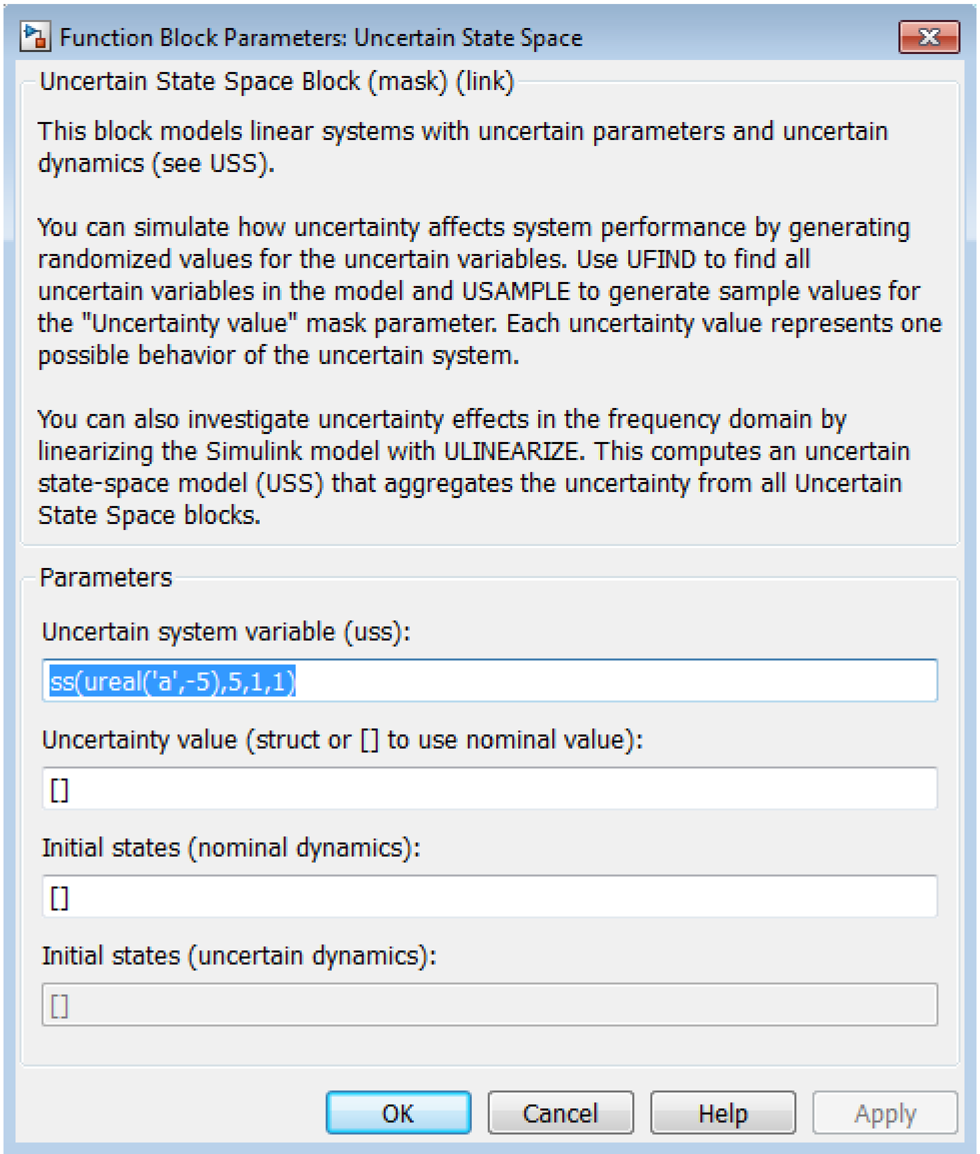
Specifying uncertainty in the Uncertain State Space block makes the uncertainty a part of the Simulink model and affects both simulation and linearization. Use this approach to vary the uncertainty and analyze the effects on simulation or linearization.

To specify uncertainty in the Uncertain State Space block:

1. Drag and drop an Uncertain State Space block from the Robust Control Toolbox block library into a Simulink model. For more information on how to open the block library, see “Simulink Blocks for Analyzing Uncertainty” on page 6-2.



2. In the Simulink model, double-click the Uncertain State Space block to open the Function Block Parameters: Uncertain State Space dialog box, as shown in the following figure.



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):

[]

3 Specify an uncertain state-space model in the **Uncertain system variable (uss)** field. The model must be an `uss` object or any other model that can be converted to `uss`, such as `umat`, `ureal`, `umargin`, and `ultidyn`. The model depends on a set of uncertain variables (`ureal`, `umargin`, or `ultidyn`) and you can specify the model as one of the following:

- Function or expression that evaluates to an `uss` model. For example, `ss(ureal('a',-5),5,1,1)`.
- Variable, defined in the MATLAB workspace. For example, `unc_sys`, where `unc_sys` is defined as `ss(ureal('a',-5),5,1,1)` in the workspace.

4 Specify values for the uncertain variables that the uncertain state-space model you specify in step 3 uses. For example, if you specify the uncertain system as `ureal('g',2)*tf(1, [ureal('tau'),1])`, then you must specify values for the uncertain variables `g` and `tau`. To do so, enter a structure with fields `g` and `tau` in the **Uncertainty value (struct or** [] **to use nominal value)** field. You can also enter `[]` to use the nominal values of the uncertain parameters `g` and `tau`.

Tip: You can also use this field to vary the uncertainty values for performing Monte Carlo simulation. For more information, see “Simulate Uncertainty Effects” on page 6-7.

5. (Optional) Specify the initial states of the nominal and uncertain dynamics in the **Initial states (nominal dynamics)** and **Initial states (uncertain dynamics)** fields, respectively.

For more information on the block parameters, see the Uncertain State Space block reference page.

Next Steps

After you specify uncertainty in Uncertain State Space blocks, you can perform one of the following:

- Simulate the model using nominal, manually-defined or random values, as described in “Simulate Uncertainty Effects” on page 6-7.
- Perform an uncertain linearization, as described in “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19.

Simulate Uncertainty Effects

How to Simulate Effects of Uncertainty

As described in “Specify Uncertainty Using Uncertain State Space Blocks” on page 6-4, the uncertain state-space model you specify in the Uncertain State Space block depends on a set of uncertain variables (`ureal`, `umargin`, or `ultidyn` objects.) You can simulate the model using nominal value of these uncertain variables. Additionally, you can sample these uncertain variables and simulate the model for various values in the uncertainty range (Monte Carlo simulation.) For more information, see “How to Vary Uncertainty Values” on page 6-7. You can view and compare the simulation results for various sample values of uncertainty using the MultiPlot Graph block.

How to Vary Uncertainty Values

There are two ways to control the uncertainty values using the **Uncertainty value (struct or** [] **to use nominal value)** field of the Uncertain State Space block parameters dialog box:

- For simple models with few uncertain variables or one Uncertain State Space block, type the value in the Uncertain State Space block itself. For more information, see “Simulate Uncertain Model at Sampled Parameter Values” on page 6-8.
- For complex models with large number of uncertain variables or Uncertain State Space blocks, use a single data structure for all uncertain variables referenced by the model. Using this approach, you can collectively control the values of all or a subset of uncertain variables and toggle between nominal and user-defined values from the MATLAB prompt. For more information, see “Vary Uncertain Values Across Multiple Uncertain Blocks” on page 6-14.

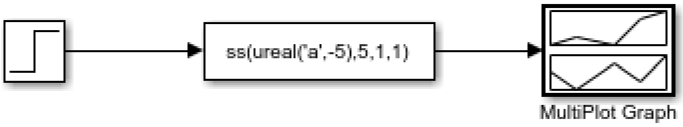
Simulate Uncertain Model at Sampled Parameter Values

This example shows how to simulate an uncertain model in Simulink® using the Uncertain State Space block. You can sample uncertain parameters at specified values or generate random samples. The MultiPlot Graph block lets you visualize the responses of multiple samples on the same plot.

Uncertain Model

The simple model `rctUncertainModel` contains an Uncertain State Space block with a step input. The step response signal feeds a MultiPlot Graph block.

```
mdl = "rctUncertainModel";
open_system(mdl)
```

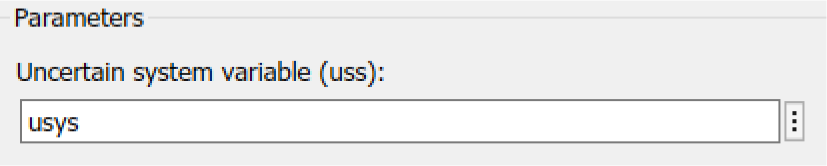


By default, the Uncertain State Space block is configured to simulate the uncertain model `ss(ureal('a',-5),5,1,1)`, which is a `uss` model with one uncertain parameter. For this example, create a model of a mass-spring damper system with an uncertain spring constant and damping constant.

```
m = 3;
c = ureal('c',1,'Percentage',20);
k = ureal('k',2,'Percentage',30);
usys = tf(1,[m c k])
```

```
usys =

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

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

To simulate this system, in the block parameters, enter `usys` for the **Uncertain system variable** parameter.


Parameters

Uncertain system variable (uss):

usys

Alternatively, set the parameter value at the command line.

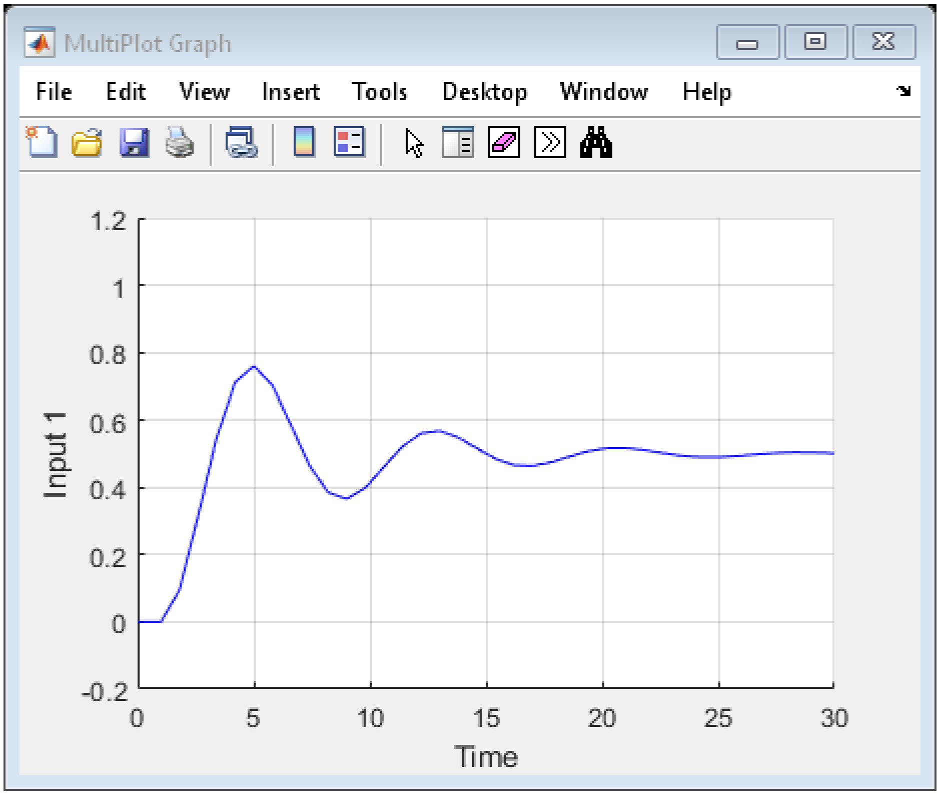
```
ublk = strcat(mdl,"/Uncertain State Space");
set_param(ublk,"USystem","usys");
```

Simulate Nominal Model

To simulate the model, Simulink must set the uncertain parameters in `usys` to specific, non-uncertain values. Use the **Uncertainty value** parameter to specify these values. By default, this parameter is set to `[]`, which causes Simulink to use the nominal values of all uncertain parameters.

Simulate the model. The MultiPlot Graph block generates a plot of the nominal model response to the step input signal.

```
sim(mdl);
```

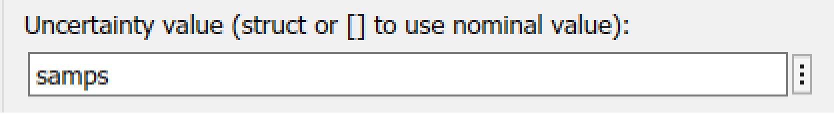


Simulate Specified Samples

To simulate the uncertain model with the uncertain parameters set to values other than the nominal values, set the **Uncertainty value** parameter to a structure whose fields are the uncertain elements of the `uss` model. For instance, create a structure `samps` that sets the damping constant to 1.2 and the spring constant to 1.7.

```
samps = struct('c',1.2,'k',1.7);
```

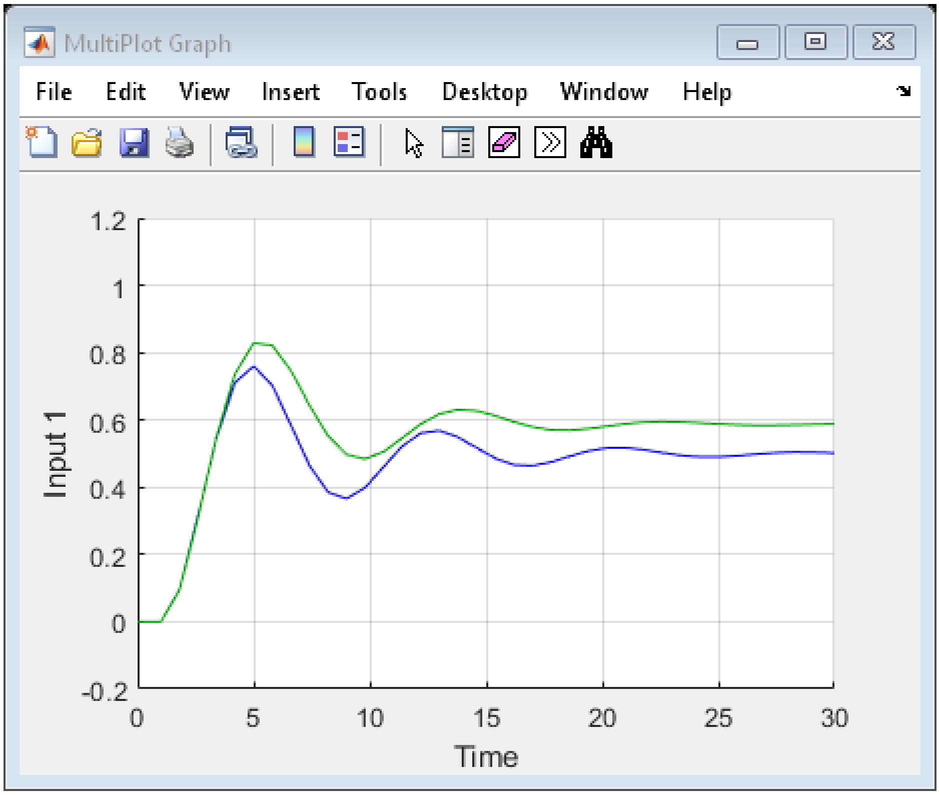
Set the **Uncertainty value** parameter to `samps`, and simulate the model. The MultiPlot Graph block adds this new system response to the same axis as the previous response.



Uncertainty value (struct or [] to use nominal value):

samps

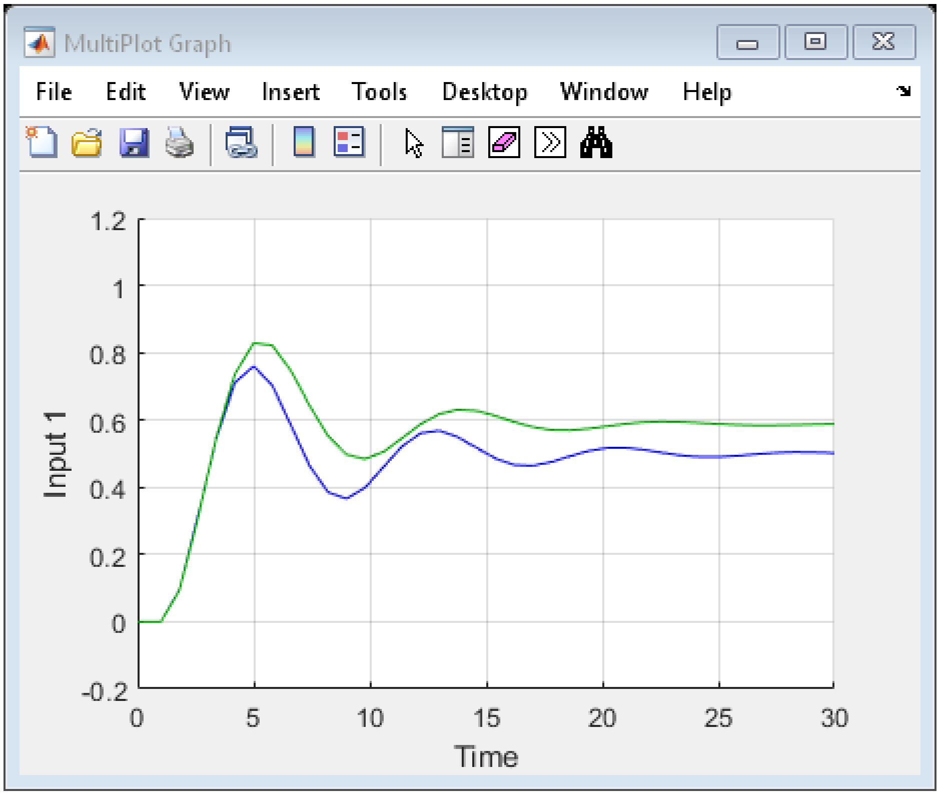
```
set_param(ublk,"UValue","samps");
sim(mdl);
```



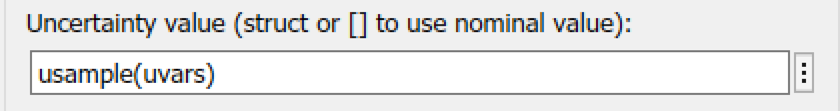
Simulate Random Samples

You can use the `usample` command to generate samples of `usys` at random values of the uncertain parameters. The command `uvars = ufind(mdl)` generates a structure containing all the uncertain parameters in the model.

```
uvars = ufind(mdl);
```

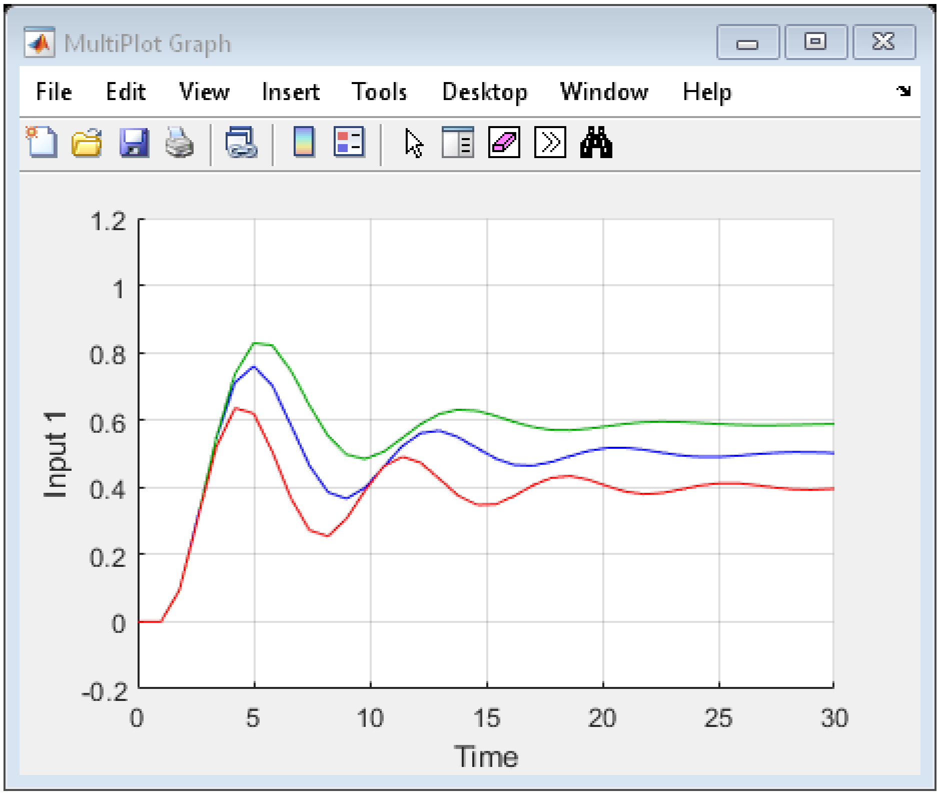


`usample` takes random samples of these parameters and returns a structure you can use for the **Uncertainty value** parameter. Set **Uncertainty value** to `usample(uvars)`, and simulate the model.


Uncertainty value (struct or [] to use nominal value):

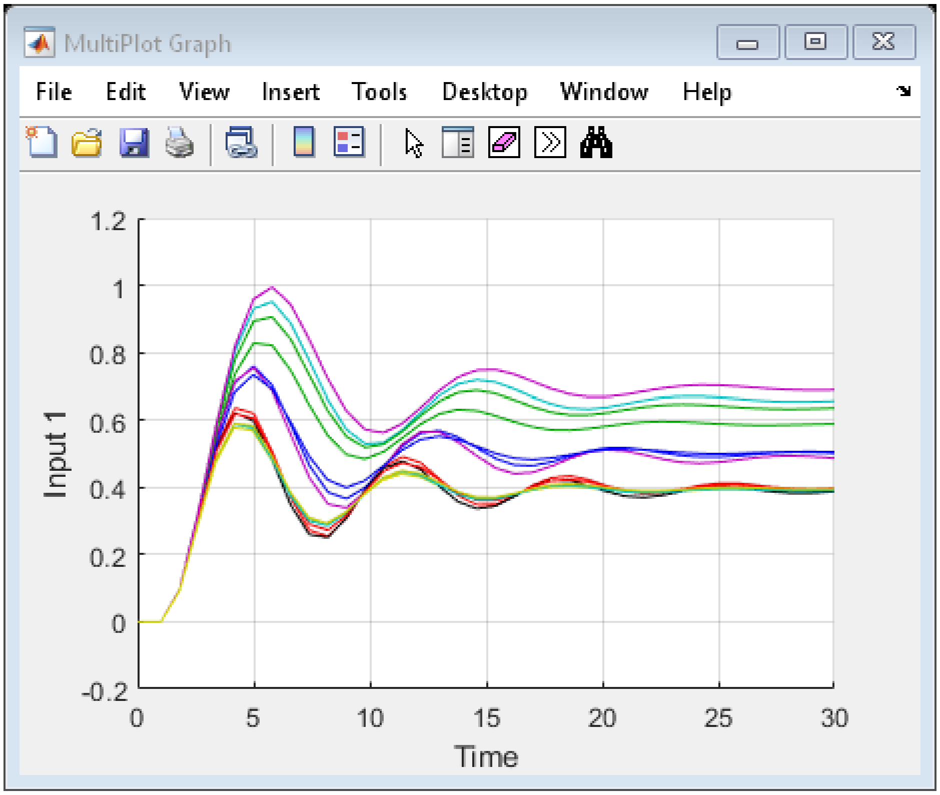
usample(uvars)

```
set_param(ublk,"UValue","usample(uvars)");
sim(mdl);
```



The step response of the randomly sampled instance of `usys` is added to the MultiPlot Graph block. Simulate the model ten more times. Each time, `usample` generates new values for `c` and `k`, and the plot is updated with another step response.

```
for i=1:10
    sim(mdl);
end
```



See Also

MultiPlot Graph | Uncertain State Space

Related Examples

- “Vary Uncertain Values Across Multiple Uncertain Blocks” on page 6-14

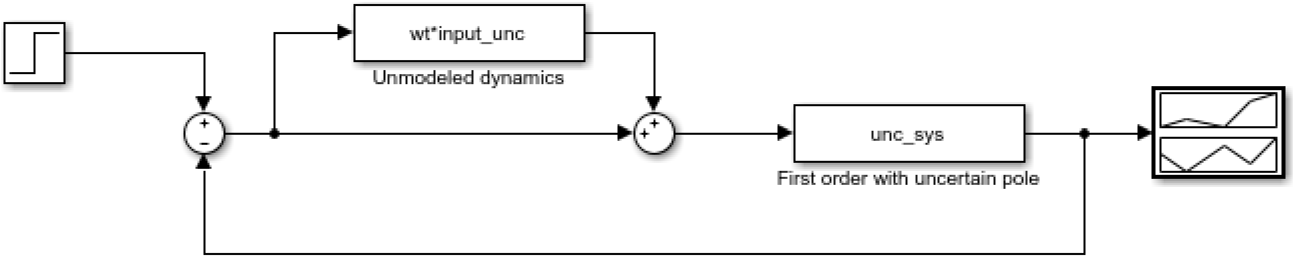
Vary Uncertain Values Across Multiple Uncertain Blocks

This example shows how to simulate a Simulink® model containing multiple Uncertain State Space blocks. You can sample all the uncertain blocks at once using the uvars command. This approach is useful when your model contains large numbers of uncertain variables or Uncertain State Space blocks.

Uncertain Model

Open the model rctMultiUncertainModel.

```
mdl = "rctMultiUncertainModel";
open_system(mdl)
```



The model is contains two Uncertain State Space blocks. The Unmodeled dynamics block is preconfigured to represent uncertain dynamics with a frequency-dependent weight of the form wt*input_unc.

```
input_unc = ultidyn('input_unc',[1 1]);
wt = makeweight(0.25,130,2.5);
```

The other uncertain block is configured to represent a first-order system with an uncertain pole location.

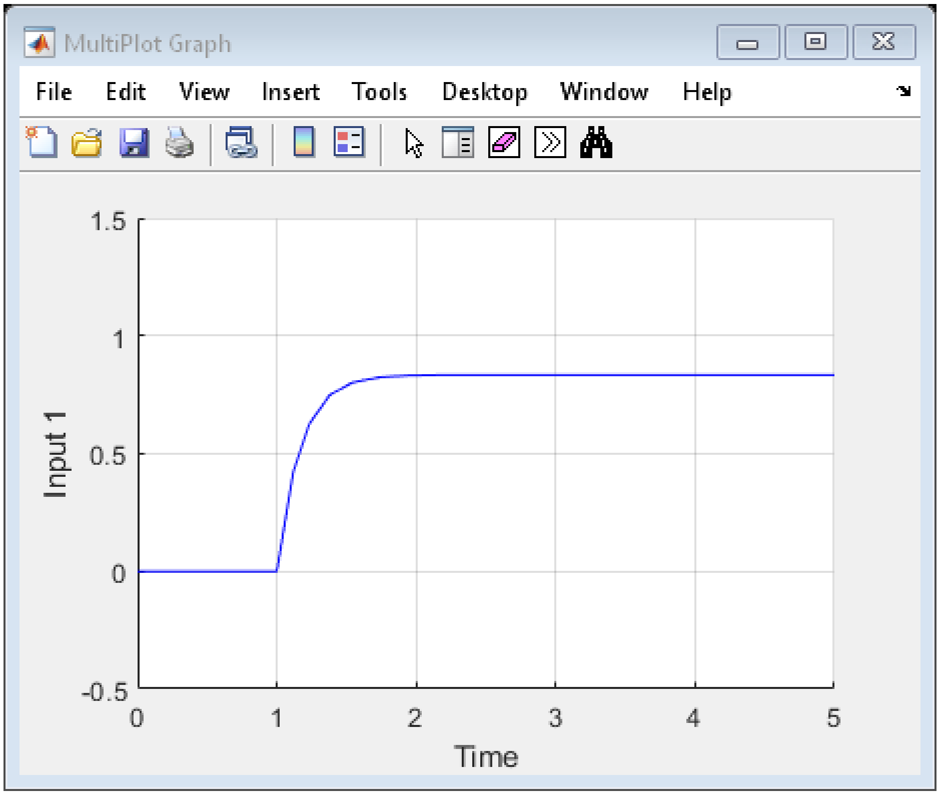
```
unc_sys = ss(ureal('a',-1,'Range',[-2 -.5]),1,5,0);
```

A step input feeds the uncertain system, and the MultiPlot Graph block shows the system.

Simulate Nominal Model

To simulate the model, Simulink must set the uncertain parameters in both of these blocks to specific, non-uncertain values. Use the **Uncertainty value** parameter to specify these values. In rctMultiUncertainModel, both blocks are preconfigured to use the workspace variable vals for that parameter. To simulate the model using the nominal values of all uncertain parameters, set vals = [].

```
vals = [];
sim(mdl);
```

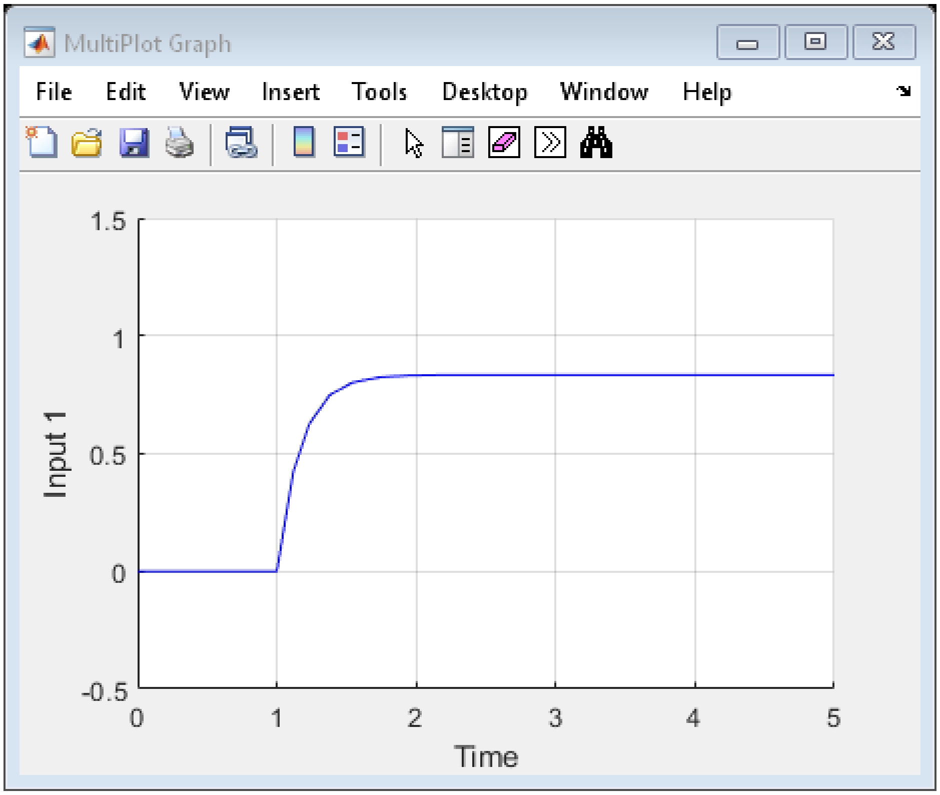


Generate Random Sample of All Uncertain Parameters

The `ufind` command finds all uncertain parameters in all the Uncertain State Space blocks across the entire model, and returns a structure containing their names and values.

```
uvars = ufind(mdl)
```

```
uvars = 

  struct with fields:

            a: [1x1 ureal]
    input_unc: [1x1 ultidyn]
```



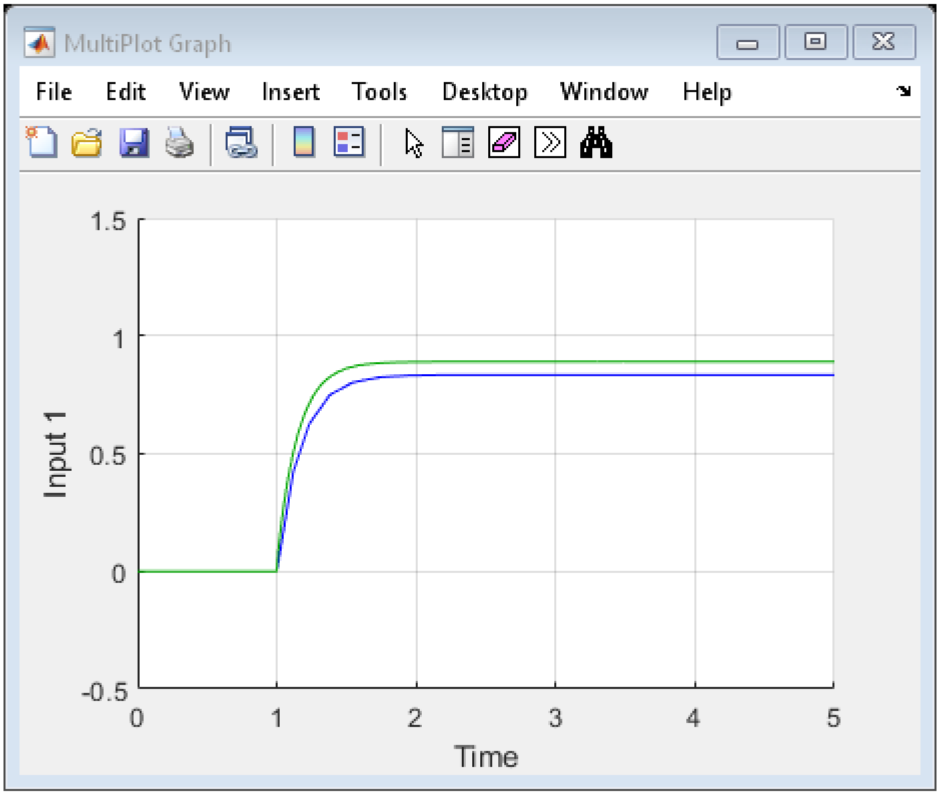
Use `usample` to generate a random sample of the uncertain parameters in uvars. Set `vals` to this sample value.

```
vals = usample(uvars)
```

```
vals = 

  struct with fields:

            a: -0.7779
    input_unc: [1x1 ss]
```

The **Uncertainty value** parameter in each Uncertain State Space block is already set to `vals`. When you simulate the model, for each block, Simulink uses the value in `vals` that corresponds to the uncertain parameters in that block.

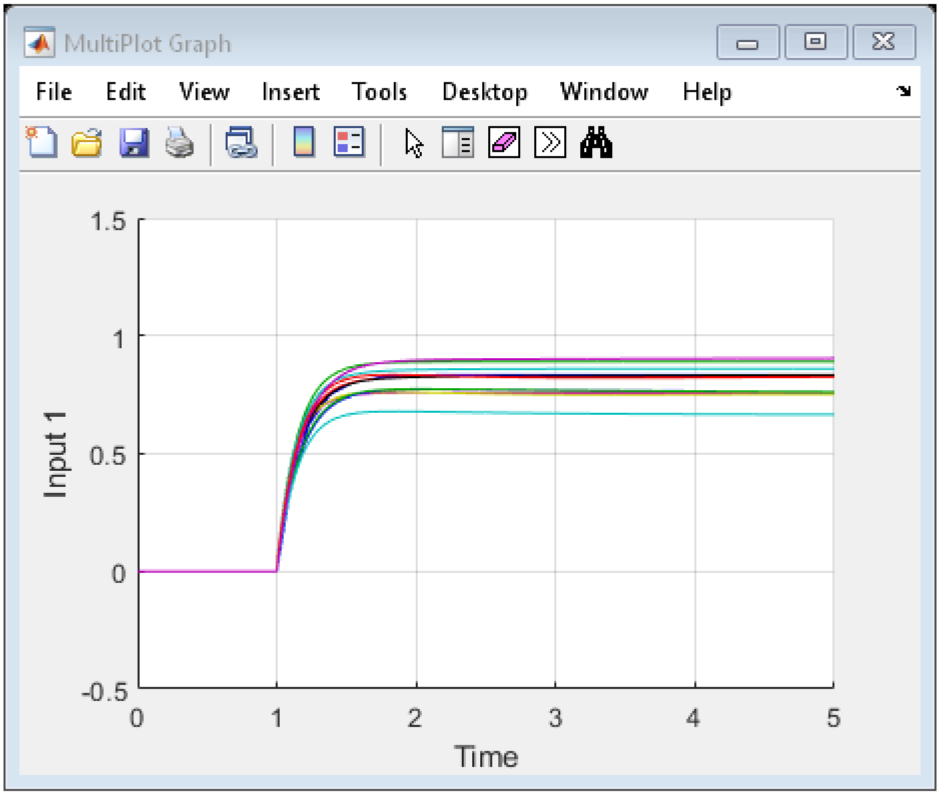
```
sim(mdl)
```



Simulate Multiple Random Samples

To simulate the model at multiple random values, repeat the process of generating random values for `vals` inside a `for` loop. Each time, `usample` generates new values for the uncertain elements in the model, and the plot is updated with another step response.

```
for i=1:10
    vals = usample(uvars);
    sim(mdl);
end
```



See Also

MultiPlot Graph | Uncertain State Space

Related Examples

- “Simulate Uncertain Model at Sampled Parameter Values” on page 6-8

Compute Uncertain State-Space Models from Simulink Models

When you have the Simulink Control Design software, you can compute an *uncertain linearization*, i.e., an uncertain state-space model (`uss`) combining the uncertain variables with linearized dynamics. Use the `uss` model to perform linear analysis and robust control design.

You can compute an uncertain linearization in one of the following ways:

- Using the `ulinearize` command, as described in “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19.
- Using the Simulink Control Design `linearize` command, as described in “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 6-19.

Obtain Uncertain State-Space Model from Simulink Model

To obtain an uncertain state-space model from a model that contains Uncertain State Space blocks, use the following steps:

Note If you do not have Uncertain State Space blocks in the model but still want to obtain an uncertain state-space model, see “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 6-19.

1. (Prerequisite) Create or open the Simulink model.
2. (Prerequisite) In the Simulink model, specify the linearization input and output points using Simulink Control Design `getlinio` or `linio` commands. For more information, see “Specify Portion of Model to Linearize” (Simulink Control Design).
3. (Prerequisite) If you have not already done so, specify uncertainty in the Simulink model as described in “Specify Uncertainty Using Uncertain State Space Blocks” on page 6-4.

   Note The software does not evaluate the uncertain variables during linearization. Thus, the value of the uncertainty does not affect the linearization.
4. Run `ulinearize` to compute an uncertain linearization. This command returns an `uss` model.

   Note If you use the Simulink Control Design`linearize` command, the Uncertain State Space blocks linearize to their nominal value.

For more information on linearization and how to evaluate the results, see “Linearization Basics” (Simulink Control Design).

For an example of how to use the Simulink Control Design `linearize` command, see “Linearization of Simulink Models with Uncertainty”.

Specify Uncertain Linearization for Core or Custom Simulink Blocks

In some cases, you cannot use Uncertain State Space blocks in the Simulink model because you share the model or generate code. You can still account for uncertainty in your linear analysis without specifying uncertainty using Uncertain State Space blocks. Robust Control Toolbox lets you specify a core or custom Simulink block to linearize to an uncertain variable. The linearization produces an uncertain state-space `uss` model. The specified uncertainty associates only with the block and does not affect the model simulation. For more information, see “Specify Linear System for Block Linearization Using MATLAB Expression” (Simulink Control Design).

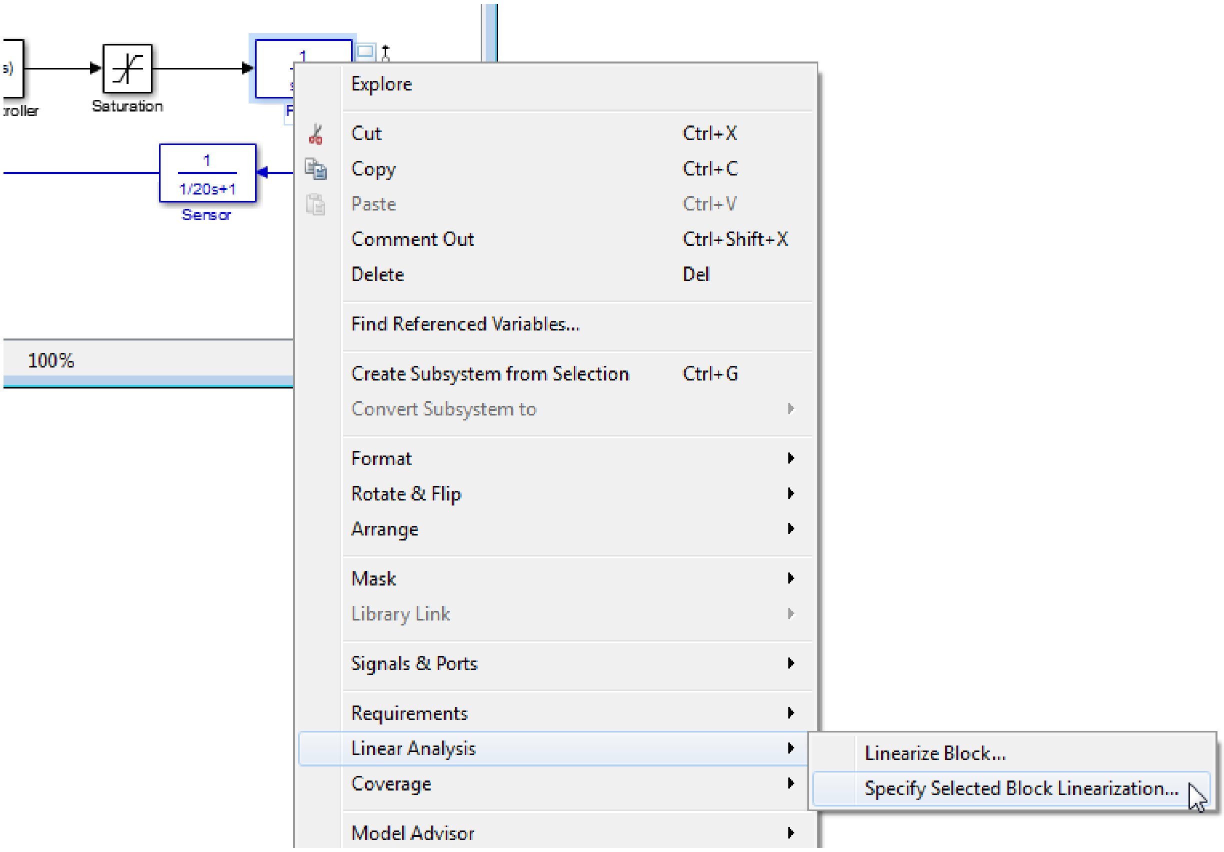
Note If you have Uncertain State Space blocks in the model and want to obtain an uncertain state-space model, see “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19.

To specify blocks to linearize to uncertain variables and obtain an uncertain state-space model:

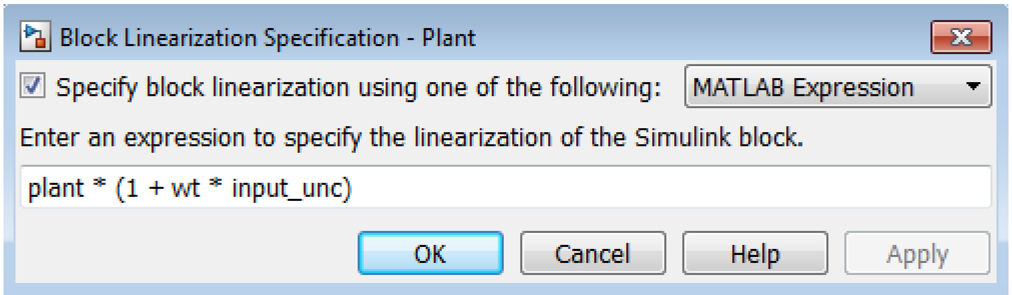
1. (Prerequisites) Create or open the Simulink model. Specify linearization input and output points using the Simulink Control Design `getlinio` or `linio` commands.

   For this example, you can open the model `rct_ulinearize_builtin`.
2. Specify a block to linearize to an uncertain variable:

   a. Right-click the block and select **Linear Analysis > Specify Selected Block Linearization.**



This action opens the Block Linearization Specification dialog box.



Block Linearization Specification - Plant

☑ Specify block linearization using one of the following: MATLAB Expression

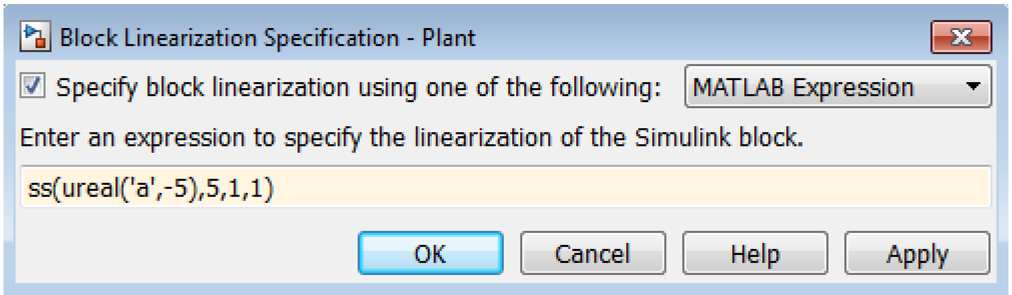
Enter an expression to specify the linearization of the Simulink block.

plant * (1 + wt * input_unc)

b In the Block Linearization Specification dialog box, select the **Specify block linearization using one of the following:** check box. Selecting this check box lets you to specify an uncertain variable for linearization.

This check box defaults to `MATLAB Expression` in the drop-down menu. This option lets you specify the block to linearize to an uncertain variable using a MATLAB expression containing Robust Control Toolbox functions. To learn more about the options, see “Specify Linear System for Block Linearization Using MATLAB Expression” (Simulink Control Design).

c In the **Enter an expression to specify the linearization of the Simulink block** field, enter an expression, which must evaluate to an uncertain variable or uncertain model, such as `ureal`, `umat`, `ultidyn`, `umargin`, or `uss`.



Block Linearization Specification - Plant

☑ Specify block linearization using one of the following: MATLAB Expression

Enter an expression to specify the linearization of the Simulink block.

ss(ureal('a',-5),5,1,1)

d Click **OK** to save the changes.

Note You can also specify a block to linearize to an uncertain variable at the command line. For an example, see “Linearize Simulink Block to Uncertain Model” on page 6-23.

3 Run the `linearize` command to compute an uncertain linearization. This command returns an `uss` model.

For more information on linearization and how to validate linearization results, see “Linearization Basics” (Simulink Control Design).

For an example of how to use the `linearize` command to compute an uncertain linearization, see “Linearization of Simulink Models with Uncertainty”.

Using Uncertain Linearization for Analysis or Control Design

After computing an uncertain linearization, you can perform any analysis or design tasks you would perform on any linear model, including:

- Perform robustness analysis. See “Robustness and Worst-Case Analysis”.
- Perform robust control design. See “Robust Controller Tuning”.

See Also

`linearize` | `ulinearize`

Related Examples

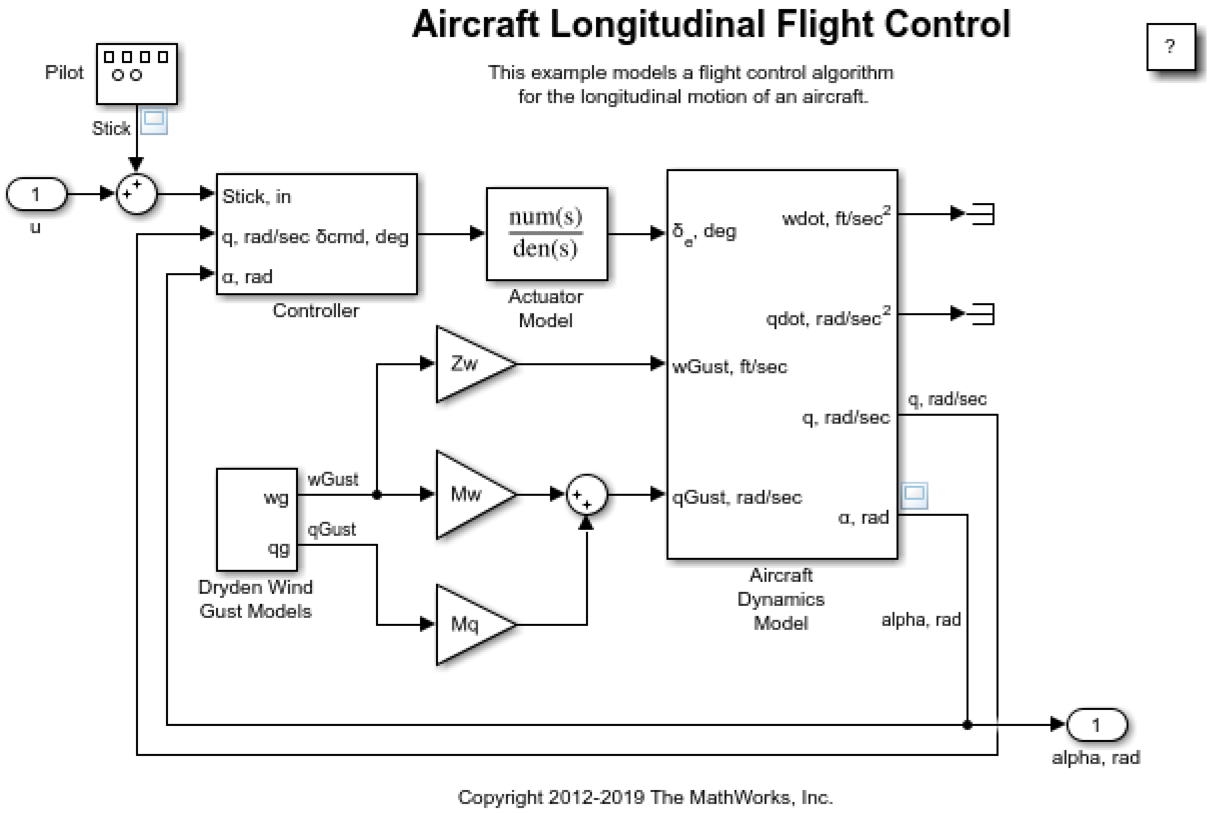
- “Linearization of Simulink Models with Uncertainty”
- “Linearize Simulink Block to Uncertain Model” on page 6-23

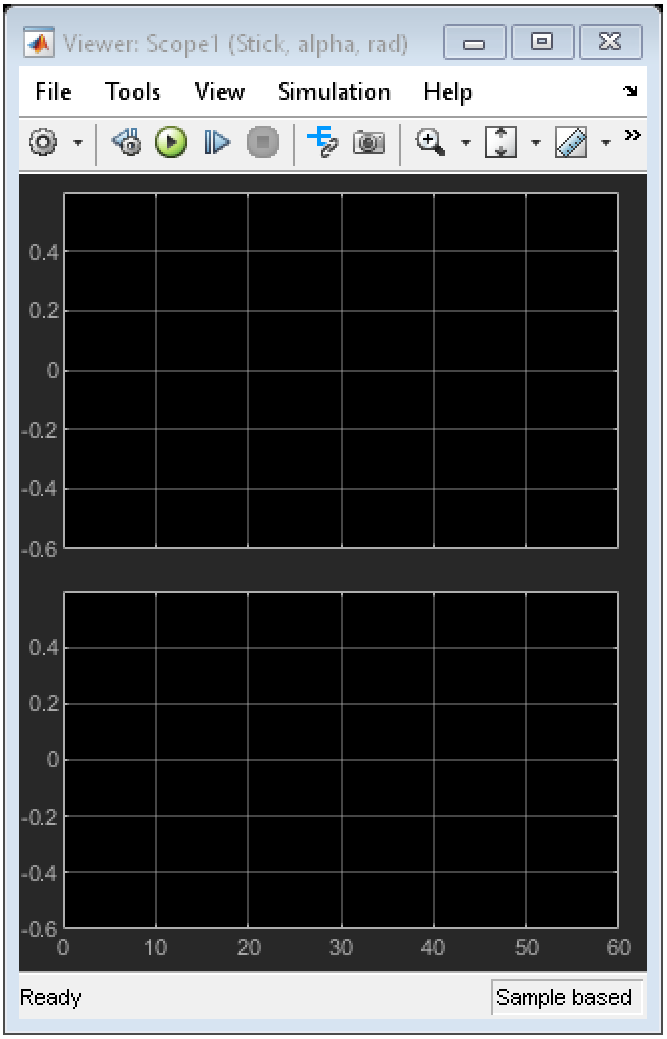
Linearize Simulink Block to Uncertain Model

This example shows how to make a Simulink® block linearize to an uncertain variable at the command line. To learn how to specify an uncertain block linearization using the Simulink model editor, see “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 6-19.

For this example, open the Simulink model `slexAircraftExample`.

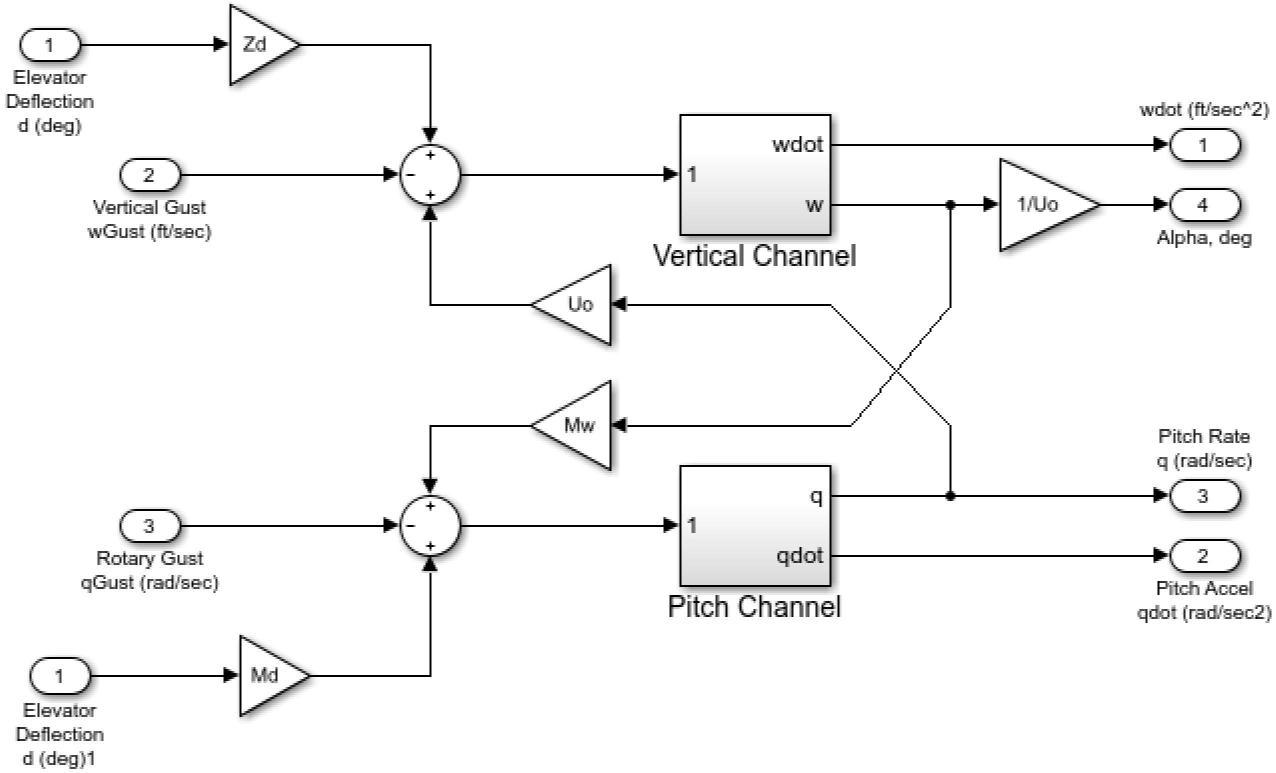
```
mdl = 'slexAircraftExample';
open_system(mdl)
```





Examine the subsystem `Aircraft Dynamics Model`.

```
subsys = [mdl,'/Aircraft Dynamics Model'];
open_system(subsys)
```



Suppose you want to specify the following uncertain real values for the gain blocks Mw and Zd.

```
Mw_unc = ureal('Mw',-0.00592,'Percentage',50);
Zd_unc = ureal('Zd',-63.9979,'Percentage',30);
```

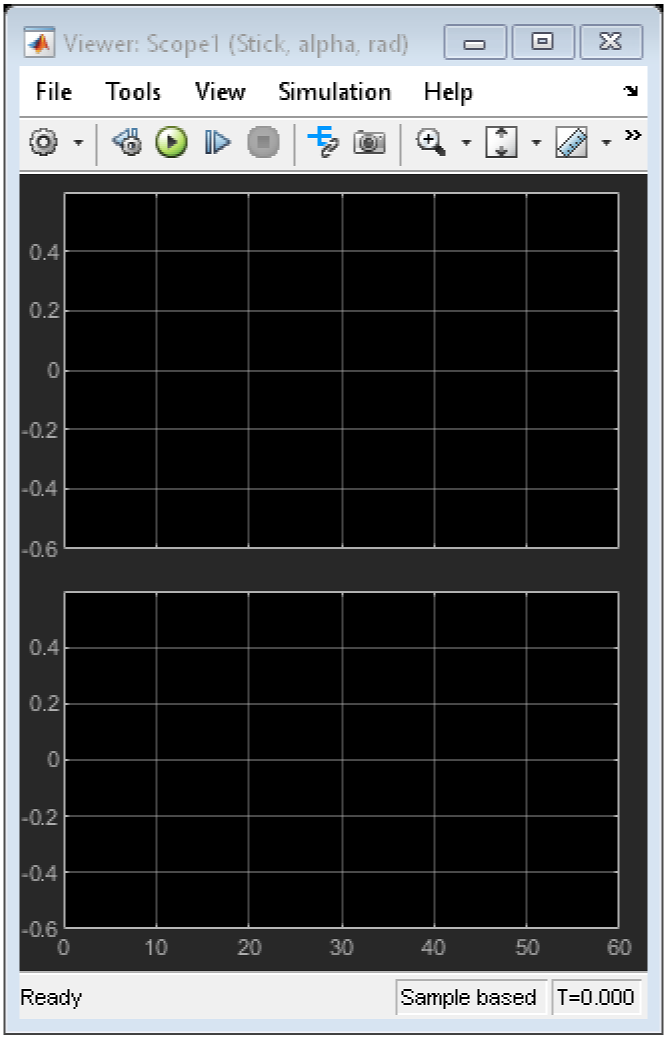
To specify these values as the linearization for these blocks, create a BlockSubs structure to pass to the linearize function. The field names are the names of the Simulink blocks, and the values are the corresponding uncertain values. Note that in this model, the name of the Mw block is Gain4, and the name of the Zd block is Gain5.

```
Mw_name = [subsys,'/Gain4'];
Zd_name = [subsys,'/Gain5'];

BlockSubs(1).Name = Mw_name;
BlockSubs(1).Value = Mw_unc;
BlockSubs(2).Name = Zd_name;
BlockSubs(2).Value = Zd_unc;
```

Compute the uncertain linearization. linearize linearizes the model at operating point specified in the model, making the substitutions specified by BlockSubs. The result is an uncertain state-space model with an uncertain real parameter for each of the two uncertain gains.

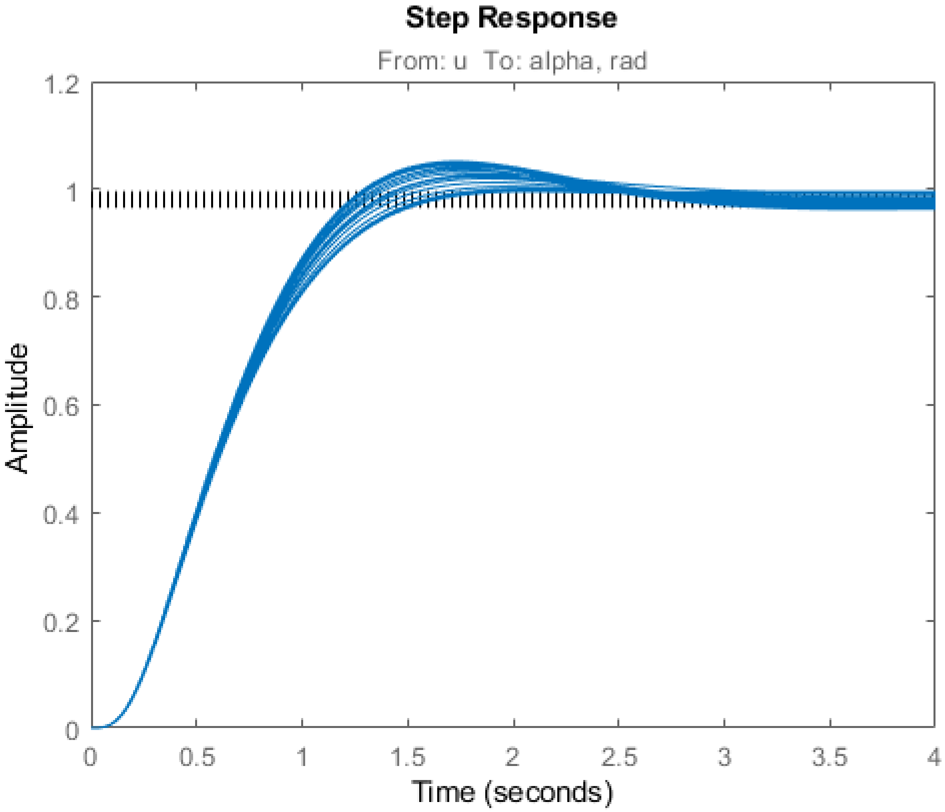
```
sys = linearize(mdl,BlockSubs)
```

```
sys =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 7 states.
  The model uncertainty consists of the following blocks:
    Mw: Uncertain real, nominal = -0.00592, variability = [-50,50]%, 1 occurrences
    Zd: Uncertain real, nominal = -64, variability = [-30,30]%, 1 occurrences

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



Examine the uncertain model response.

```
step(sys)
```



`step` takes random samples and provides a sense of the range of responses within the uncertainty of the linearized model.

See Also

`linearize`

Related Examples

- “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 6-19
- “Obtain Uncertain State-Space Model from Simulink Model” on page 6-19

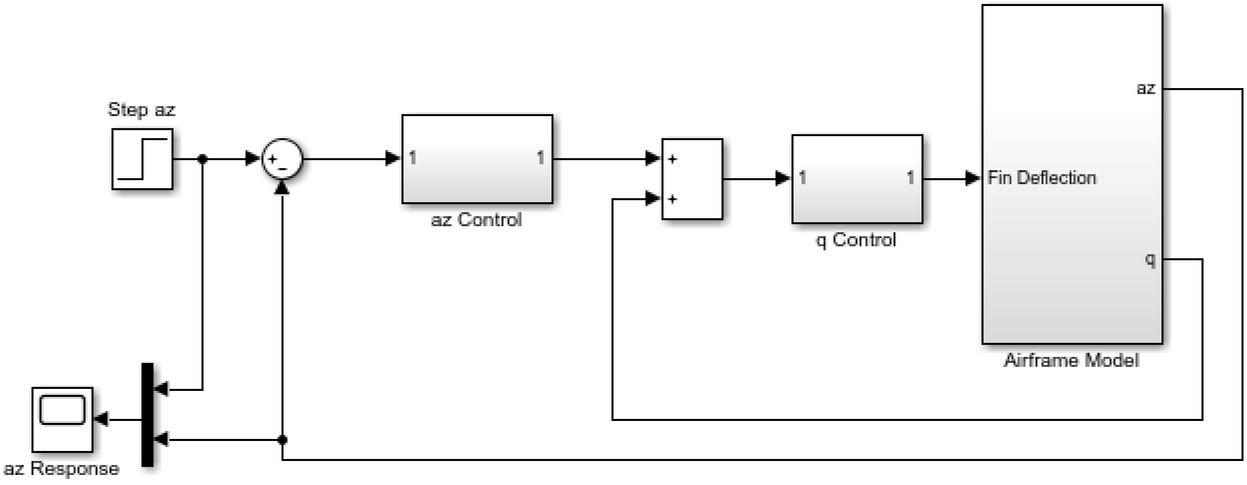
Stability Margins of a Simulink Model

This example illustrates how to compute classical and disk-based gain and phase margins of a control loop modeled in Simulink®. To compute stability margins, linearize the model to extract the open-loop responses at one or more operating points of interest. Then, use `allmargin` or `diskmargin` to compute the classical or disk-based stability margins, respectively.

MIMO Control Loop

For this example, use the Simulink model `airframemarginEx.slx`. This model is based on the example “Trimming and Linearizing an Airframe” (Simulink Control Design).

```
open_system('airframemarginEx.slx')
```



Copyright 2004-2014 The MathWorks, Inc.

The system is a two-channel feedback loop. The plant is the one-input, two-output subsystem `Airframe Model`, and the controller is a two-input, one-output system whose inputs are the normal acceleration `az` and pitch rate `q`, and whose output is the `Fin Deflection` signal.

Loop Transfer Functions

To compute the gain margins and phase margins for this feedback system, linearize the model to get the open-loop transfer functions at the plant outputs and input. You can do so using linearization analysis points of the loop-transfer type. For more information about linearization analysis points, see “Specify Portion of Model to Linearize” (Simulink Control Design).

Create a loop-transfer analysis point for the plant input, which is the first output port of the `q Control` subsystem.

```
ioInput = linio('airframemarginEx/q Control',1,'looptransfer');
```

Similarly, create analysis points for the plant outputs. Because there are two outputs, specify these analysis points as a vector of linearization I/O objects.

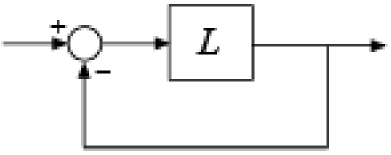
```
ioOutput(1) = linio('airframemarginEx/Airframe Model',1,'looptransfer');
ioOutput(2) = linio('airframemarginEx/Airframe Model',2,'looptransfer');
```

Linearize the model to obtain the open-loop transfer functions. For this example, use the operating point specified in the model. The loop transfer at the plant input is SISO, while the loop transfer at the outputs is 2-by-2.

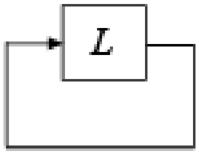
```
Li = linearize('airframemarginEx',ioInput);   % SISO
Lo = linearize('airframemarginEx',ioOutput);  % MIMO
```

Classical Gain and Phase Margins

To compute the classical gain margins and phase margins, use `allmargin`. For an open-loop transfer function, `allmargin` assumes a negative-feedback loop.



The open-loop transfer function returned by the `linearize` command is the actual linearized open-loop response of the model at the analysis point. Thus, for an open-loop response `L`, the closed-loop response of the entire model is a positive feedback loop.



Therefore, use `-L` to make `allmargin` compute the stability margins with positive feedback. Compute the classical gain and phase margins at the plant input.

```
Si = allmargin(-Li)
```

```
Si = 

  struct with fields:

     GainMargin: [0.1633 17.6572]
    GMFrequency: [1.5750 47.5284]
    PhaseMargin: 44.4554
    PMFrequency: 5.3930
    DelayMargin: 14.3869
    DMFrequency: 5.3930
         Stable: 1
```

The structure `Si` contains information about classical stability margins. For instance, `Li.GMFrequency` gives the two frequencies at which the phase of the open-loop response crosses –180°. `Li.GainMargin` gives the gain margin at each of those frequencies. The gain margin is the amount by which the loop gain can vary at that frequency while preserving closed-loop stability.

Compute the stability margins at the plant output.

```
So = allmargin(-Lo);
```

Because there are two output channels, `allmargin` returns an array containing one structure for each channel. Each entry contains the margins computed for that channel with the other feedback channel closed. Index into the structure `So` to obtain the stability margins for each channel. For instance, examine the margins with respect to gain variations or phase variations at the `q` output of the plant, which is the second output.

```
So(2)
```

```
ans = 

  struct with fields:

     GainMargin: [0.3456 17.4301]
    GMFrequency: [3.4362 49.8484]
    PhaseMargin: [-78.2436 52.6040]
    PMFrequency: [1.5686 6.5428]
    DelayMargin: [313.5079 14.0324]
    DMFrequency: [1.5686 6.5428]
         Stable: 1
```

Disk-Based Gain and Phase Margins

Disk margins provide a stronger guarantee of stability than the classical gain and phase margins. Disk-based margin analysis models gain and phase variations as a complex uncertainty on the open-loop system response. The disk margin is the smallest such uncertainty that is compatible with closed-loop stability. (For general information about disk margins, see “Stability Analysis Using Disk Margins” on page 2-2.)

To compute disk-based margins, use `diskmargin`. Like `allmargin`, the `diskmargin` command assumes a negative-feedback system. Thus, use `-Li` to compute the disk-based margins at the plant input.

```
DMi = diskmargin(-Li)
```

```
DMi = 

  struct with fields:

           GainMargin: [0.4419 2.2628]
          PhaseMargin: [-42.3153 42.3153]
           DiskMargin: 0.7740
           LowerBound: 0.7740
           UpperBound: 0.7740
            Frequency: 4.2515
    WorstPerturbation: [1x1 ss]
```

The field `DMi.GainMargin` tells you that the open-loop gain at the plant input can vary by any factor between about 0.44 and about 2.26 without loss of closed-loop stability. Disk-based margins take into account variations at all frequencies.

For a MIMO loop transfer function such as the response `Lo` at the plant outputs, there are two types of disk-based stability margins. The *loop-at-a-time margins* are the stability margins in each channel with the other loop closed. The *multiloop margins* are the margins for independent variations in gain (or phase) in both channels simultaneously. `diskmargin` computes both.

```
[DMo,MMo] = diskmargin(-Lo);
```

The loop-at-a-time margins are returned as a structure array `DMo` with one entry for each channel. For instance, examine the margins for gain variations or phase variations at the `q` output of the plant with the `az` loop closed, and compare with the classical margins given by `So(2)` above.

```
DMo(2)
```

```
ans = 

  struct with fields:

           GainMargin: [0.3771 2.6521]
          PhaseMargin: [-48.6811 48.6811]
           DiskMargin: 0.9047
           LowerBound: 0.9047
           UpperBound: 0.9047
            Frequency: 4.4982
    WorstPerturbation: [2x2 ss]
```

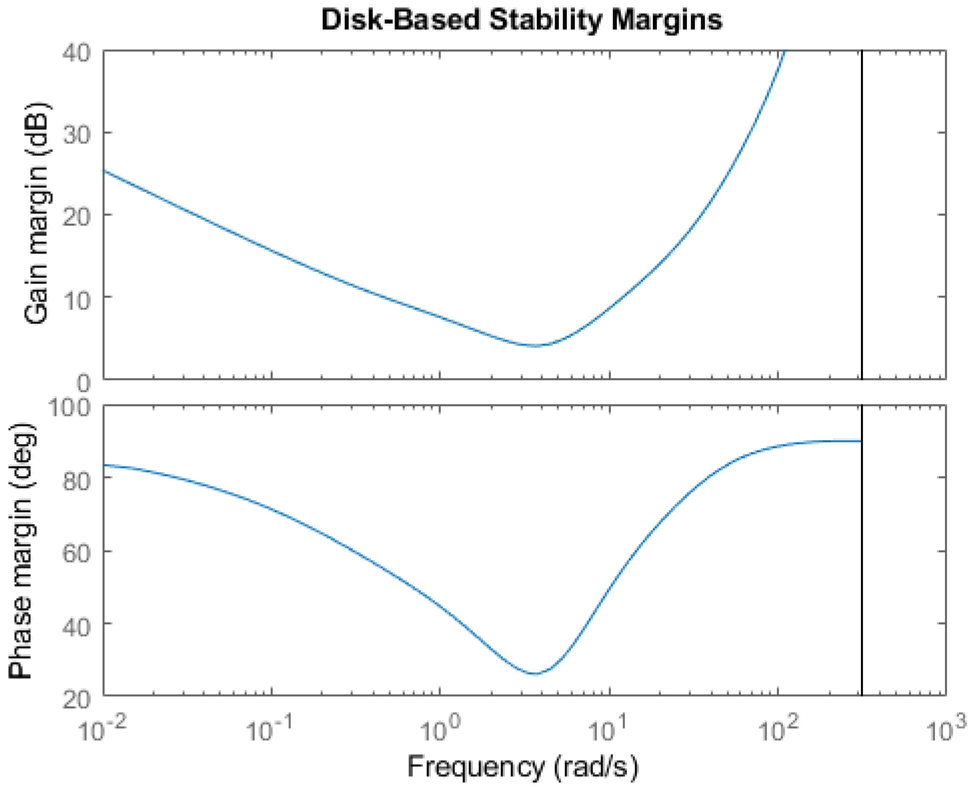
The multiloop margin, `MMo`, takes into account loop interaction by considering simultaneous variations in gain (or phase) across all feedback channels. This typically gives the most realistic stability margin estimate for multiloop control systems.

```
MMo
```

```
MMo = 

  struct with fields:

           GainMargin: [0.6238 1.6030]
          PhaseMargin: [-26.0867 26.0867]
           DiskMargin: 0.4633
           LowerBound: 0.4633
           UpperBound: 0.4643
            Frequency: 3.6830
    WorstPerturbation: [2x2 ss]
```

`MMo.GainMargin` shows that the gains in both output channels can vary independently by factors between about 0.62 and 1.60 without compromising closed-loop stability. `MMo.PhaseMargin` shows that stability is preserved for independent phase variations in each channel of up to ±26°. Use `diskmarginplot` to examine the multiloop margins graphically.

```
diskmarginplot(-Lo)
```



This shows the disk-based gain and phase margins as a function of frequency. The `MMo` values returned by `diskmargin` correspond to the weakest disk margin across frequency.

Margins at Multiple Operating Points

When you use `linearize`, you can provide multiple operating points to generate an array of linearizations of the system. `allmargin` and `diskmargin` can operate on linear model arrays to return the margins at multiple operating points. For example, linearize the airframe system at three simulation snapshot times.

```
Snap = [0; 2; 5];
LiSnap = linearize('airframemarginEx',ioInput,Snap);
LoSnap = linearize('airframemarginEx',ioOutput,Snap);
```

`LiSnap` is a 3-by-1 array of SISO linear models, one for the loop transfer at the plant input obtained at each snapshot time. Similarly, `LoSnap` is a 3-by-1 array of 2-input, 2-output linear models representing the loop transfers at the plant outputs at each snapshot time. Compute the classical gain and phase margins at the plant inputs at the three snapshot times.

```
SiSnap = allmargin(-LiSnap);
```

Each entry in the structure array `SiSnap` contains the classical margin information for the corresponding snapshot time. For instance, examine the classical margins for the second entry, `t` = 2 s.

```
SiSnap(2)
```

```
ans = 

  struct with fields:

     GainMargin: [0.0171 18.2489]
    GMFrequency: [0.0502 51.4426]
    PhaseMargin: 93.1051
    PMFrequency: 2.8476
    DelayMargin: 57.0662
    DMFrequency: 2.8476
         Stable: 1
```

Compute the disk margins at the plant outputs.

```
[DMoSnap,MMoSnap] = diskmargin(-LoSnap);
```

Because there are two feedback channels and three snapshot times, the structure array containing the loop-at-a-time disk margins has dimensions 2-by-3. The first dimension is for the feedback channels, and the second is for the snapshot times. In other words, `DMoSnap(j,k)` contains the margins for the channel `j` at the snapshot time `k`. For instance, examine the disk margins in the second feedback channel at the third snapshot time, `t` = 5 s.

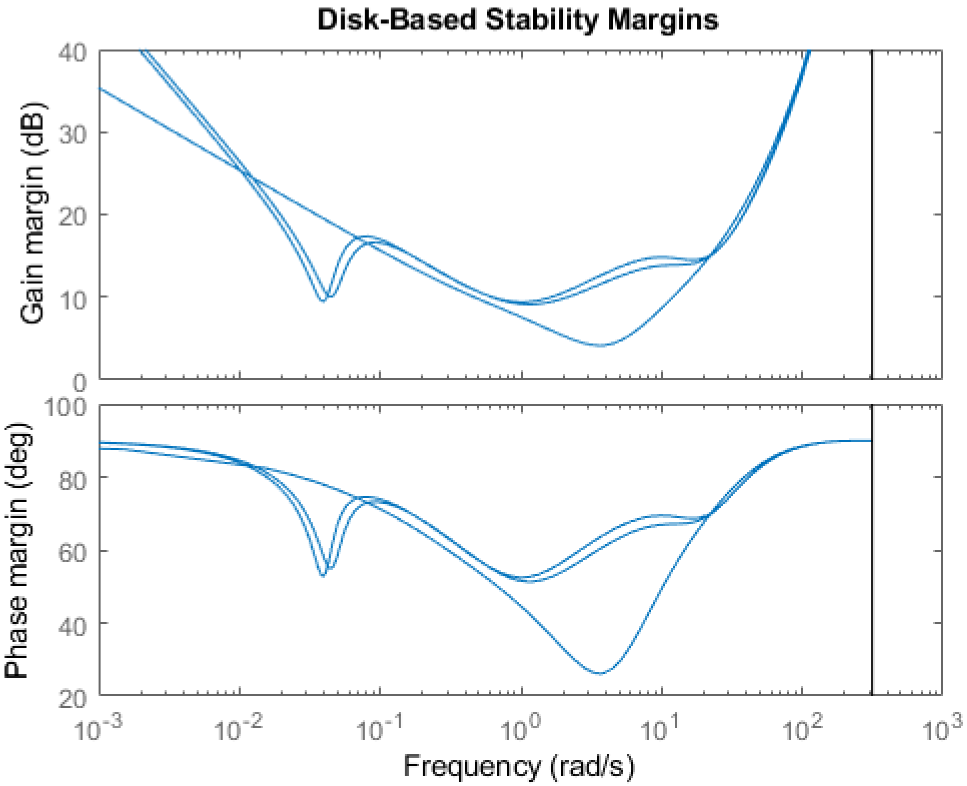
```
DMoSnap(2,3)
```

```
ans = 

  struct with fields:

           GainMargin: [0.1345 7.4338]
          PhaseMargin: [-74.6771 74.6771]
           DiskMargin: 1.5257
           LowerBound: 1.5257
           UpperBound: 1.5257
            Frequency: 24.1993
    WorstPerturbation: [2x2 ss]
```

There is only one set of multiloop margins for each snapshot time, so `MMoSnap` is a 3-by-1 structure array.

As before, you can also plot the multiloop margins. There are now three curves, one for each snapshot time. Click on a curve to identify which snapshot time it corresponds to.

```
diskmarginplot(-LoSnap)
```



See Also

`allmargin` | `diskmargin` | `diskmarginplot` | `linearize`

More About

- “Stability Analysis Using Disk Margins” on page 2-2