### Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2005</td>
<td>First printing</td>
<td>New for Version 3.0.2 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for Version 3.1 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for Version 3.1.1 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for Version 3.2 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Version 3.3 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Version 3.3.1 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Version 3.3.2 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Version 3.3.3 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for Version 3.4 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for Version 3.4.1 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Version 3.5 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Version 3.6 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Version 4.0 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Version 4.1 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Version 4.2 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Version 4.3 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Version 5.0 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Version 5.1 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Version 5.2 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Version 5.3 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Version 6.0 (Release 2015b)</td>
</tr>
</tbody>
</table>
Uncertain State-Space Models (uss) ........................................ 1-33
Properties of uss Objects .................................................. 1-33

Create Uncertain State-Space Model ................................. 1-36

Sample Uncertain Systems ............................................. 1-37

Feedback Around an Uncertain Plant .............................. 1-39

Interpreting Uncertainty in Discrete Time ......................... 1-41

Lifting a ss to a uss ..................................................... 1-42

Create Uncertain Frequency Response Data Models ............ 1-43
Properties of ufrd Objects ............................................. 1-44

Interpreting Uncertainty in Discrete Time ......................... 1-46

Lifting an frd to a ufrd .................................................. 1-47

Basic Control System Toolbox and MATLAB
Interconnections ......................................................... 1-48

Simplifying Representation of Uncertain Objects ............... 1-49
Effect of the Autosimplify Property ............................... 1-50
Direct Use of simplify .................................................. 1-52

Generate Samples of Uncertain Systems ......................... 1-53
Generating One Sample ................................................. 1-53
Generating Many Samples ............................................. 1-53
Sampling ultidyn Elements ............................................ 1-54

Substitution by usubs .................................................. 1-57
Specifying the Substitution with Structures .................... 1-58
Nominal and Random Values ........................................... 1-58

Array Management for Uncertain Objects ....................... 1-60

Reference Into Arrays .................................................. 1-61

Create Arrays with stack and cat Functions .................... 1-62

Create Arrays by Assignment ......................................... 1-64
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Operations with Arrays</td>
<td>1-65</td>
</tr>
<tr>
<td>Create Arrays with usample</td>
<td>1-66</td>
</tr>
<tr>
<td>Create Arrays with usubs</td>
<td>1-68</td>
</tr>
<tr>
<td>Create Arrays with gridureal</td>
<td>1-69</td>
</tr>
<tr>
<td>Create Arrays with repmat</td>
<td>1-70</td>
</tr>
<tr>
<td>Create Arrays with repsys</td>
<td>1-71</td>
</tr>
<tr>
<td>Using permute and ipermute</td>
<td>1-72</td>
</tr>
<tr>
<td>Decomposing Uncertain Objects</td>
<td>1-73</td>
</tr>
<tr>
<td>Normalizing Functions for Uncertain Elements</td>
<td>1-73</td>
</tr>
<tr>
<td>Properties of the Decomposition</td>
<td>1-74</td>
</tr>
<tr>
<td>Syntax of lftdata</td>
<td>1-75</td>
</tr>
<tr>
<td>Advanced Syntax of lftdata</td>
<td>1-77</td>
</tr>
<tr>
<td>Generalized Robustness Analysis</td>
<td>2-2</td>
</tr>
<tr>
<td>Generalized Robustness Analysis</td>
<td>2-2</td>
</tr>
<tr>
<td>Robust Stability Margin</td>
<td>2-4</td>
</tr>
<tr>
<td>Robust Performance Margin</td>
<td>2-5</td>
</tr>
<tr>
<td>Worst-Case Gain Measure</td>
<td>2-6</td>
</tr>
<tr>
<td>Introduction to Linear Matrix Inequalities</td>
<td>3-2</td>
</tr>
<tr>
<td>Linear Matrix Inequalities</td>
<td>3-2</td>
</tr>
<tr>
<td>LMI Features</td>
<td>3-2</td>
</tr>
</tbody>
</table>
LMIs and LMI Problems ........................................... 3-4
LMI Applications ......................................................... 3-6
  Stability ................................................................. 3-7
  RMS Gain ............................................................... 3-8
  LQG Performance .................................................... 3-8
Further Mathematical Background .............................. 3-10
Bibliography ............................................................. 3-11

LMI Lab

Tools for Specifying and Solving LMIs ......................... 4-2
  Some Terminology ...................................................... 4-2
  Overview of the LMI Lab .......................................... 4-5

Specifying a System of LMIs ....................................... 4-7

Specify LMI System at the Command Line ...................... 4-8
  Specifying LMI System .............................................. 4-8
  Initializing the LMI System ....................................... 4-9
  Specifying the LMI Variables ................................... 4-10
  Specifying Individual LMIs ...................................... 4-12

Specify LMIs with the LMI Editor GUI ......................... 4-15
  Keyboard Shortcuts ............................................... 4-17
  Limitations .......................................................... 4-17

How Imlivar and Imiterm Manage LMI Representation ...... 4-18

Querying the LMI System Description ......................... 4-19
  lmiinfo ............................................................... 4-19
  lminbr and matnbr .................................................. 4-19

LMI Solvers ............................................................ 4-20

Minimize Linear Objectives under LMI Constraints .......... 4-22
Conversion Between Decision and Matrix Variables ............................. 4-26

Validating Results .............................................................................. 4-27

Modify a System of LMIs ................................................................. 4-28
  Deleting an LMI .............................................................................. 4-28
  Deleting a Matrix Variable ............................................................. 4-28
  Instantiating a Matrix Variable ....................................................... 4-29

Advanced LMI Techniques ............................................................... 4-31
  Structured Matrix Variables ......................................................... 4-31
  Complex-Valued LMIs .................................................................... 4-33
  Specifying $c^T x$ Objectives for mincx ........................................ 4-36
  Feasibility Radius .......................................................................... 4-37
  Well-Posedness Issues .................................................................... 4-38
  Semi-Definite $B(x)$ in gevp Problems ......................................... 4-39
  Efficiency and Complexity Issues .................................................. 4-40
  Solving $M + P^T X Q + Q^T X^T P < 0$ ......................................... 4-40

Bibliography ...................................................................................... 4-42

5

Analyzing Uncertainty Effects in Simulink

Analyzing Uncertainty in Simulink .................................................... 5-2

Simulink Blocks for Analyzing Uncertainty ....................................... 5-3

Specify Uncertainty Using Uncertain State Space Blocks .......... 5-4
  How to Specify Uncertainty in Uncertain State Space Blocks ........ 5-4
  Next Steps ...................................................................................... 5-6

Simulate Uncertainty Effects ............................................................ 5-7
  How to Simulate Effects of Uncertainty ........................................ 5-7
  How to Vary Uncertainty Values ..................................................... 5-7

Vary Uncertainty Values Using Individual Uncertain State
  Space Blocks .................................................................................. 5-8
Vary Uncertainty Values Across Multiple Uncertain State Space Blocks ................................................................. 5-14

Computing Uncertain State-Space Models from Simulink Models ............................................................................... 5-19

Obtain Uncertain State-Space Model from Simulink Model ......................................................................................... 5-20

Specify Uncertain Linearization for Core or Custom Simulink Blocks ........................................................................ 5-21

Linearize Block to Uncertain Model .................................................. 5-24

Using Uncertain Linearization for Analysis or Control Design .................................................................................... 5-26

Analyzing Stability Margin of Simulink Models ......................... 5-27
  How Stability Margin Analysis Using Loopmargin Differs Between Simulink and LTI Models ......................... 5-27
  Stability Margin of Simulink Model ............................................. 5-27

Stability Margin of a Simulink Model ........................................... 5-29
Building Uncertain Models

- “Introduction to Uncertain Elements” on page 1-3
- “Uncertain Real Parameters” on page 1-5
- “Create Uncertain Real Parameters” on page 1-7
- “Uncertain LTI Dynamics Elements” on page 1-14
- “Create Uncertain LTI Dynamics” on page 1-17
- “Uncertain Complex Parameters and Matrices” on page 1-18
- “Systems with Unmodeled Dynamics” on page 1-22
- “Uncertain Matrices” on page 1-23
- “Create Uncertain Matrices from Uncertain Elements” on page 1-24
- “Access Properties of a umat” on page 1-25
- “Row and Column Referencing” on page 1-27
- “Matrix Operation on umat Objects” on page 1-29
- “Evaluate Uncertain Elements by Substitution” on page 1-31
- “Uncertain State-Space Models (uss)” on page 1-33
- “Create Uncertain State-Space Model” on page 1-36
- “Sample Uncertain Systems” on page 1-37
- “Feedback Around an Uncertain Plant” on page 1-39
- “Interpreting Uncertainty in Discrete Time” on page 1-41
- “Lifting a ss to a uss” on page 1-42
- “Create Uncertain Frequency Response Data Models” on page 1-43
- “Interpreting Uncertainty in Discrete Time” on page 1-46
- “Lifting an frd to a ufrd” on page 1-47
- “Basic Control System Toolbox and MATLAB Interconnections” on page 1-48
- “Simplifying Representation of Uncertain Objects” on page 1-49
- “Generate Samples of Uncertain Systems” on page 1-53
• “Substitution by usubs” on page 1-57
• “Array Management for Uncertain Objects” on page 1-60
• “Reference Into Arrays” on page 1-61
• “Create Arrays with stack and cat Functions” on page 1-62
• “Create Arrays by Assignment” on page 1-64
• “Binary Operations with Arrays” on page 1-65
• “Create Arrays with usample” on page 1-66
• “Create Arrays with usubs” on page 1-68
• “Create Arrays with gridureal” on page 1-69
• “Create Arrays with repmat” on page 1-70
• “Create Arrays with repsys” on page 1-71
• “Using permute and ipermute” on page 1-72
• “Decomposing Uncertain Objects” on page 1-73
Introduction to Uncertain Elements

Uncertain elements (also called uncertain Control Design Blocks) are the building blocks used to form uncertain matrix objects and uncertain system objects. There are 5 classes of uncertain elements:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ureal</td>
<td>Uncertain real parameter</td>
</tr>
<tr>
<td>ultidyn</td>
<td>Uncertain, linear, time-invariant dynamics</td>
</tr>
<tr>
<td>ucomplex</td>
<td>Uncertain complex parameter</td>
</tr>
<tr>
<td>ucomplexm</td>
<td>Uncertain complex matrix</td>
</tr>
<tr>
<td>udyn</td>
<td>Uncertain dynamic system</td>
</tr>
</tbody>
</table>

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

```matlab
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

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

For `ureal`, `ultidyn`, `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,

```matlab
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 1-53 to learn more about usample.
Uncertain Real Parameters

An uncertain real parameter is the Control Design Block that represents a real number whose value is uncertain. 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:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Meaning</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Internal name</td>
<td>char</td>
</tr>
<tr>
<td>NominalValue</td>
<td>Nominal value of element</td>
<td>double</td>
</tr>
<tr>
<td>Mode</td>
<td>Signifies which description (from 'PlusMinus', 'Range', 'Percentage') of uncertainty is invariant when NominalValue is changed</td>
<td>char</td>
</tr>
<tr>
<td>PlusMinus</td>
<td>Additive variation</td>
<td>scalar or 1x2 double</td>
</tr>
<tr>
<td>Range</td>
<td>Numerical range</td>
<td>1x2 double</td>
</tr>
<tr>
<td>Percentage</td>
<td>Additive variation (% of absolute value of nominal)</td>
<td>scalar or 1x2 double</td>
</tr>
<tr>
<td>AutoSimplify</td>
<td>'off'</td>
<td>{'basic'}</td>
</tr>
</tbody>
</table>

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 1-49 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).

```matlab
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.

```matlab
get(a)
```

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

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.

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

```
b =

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

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

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

Create an uncertain parameter with an asymmetric variation about its nominal value. Examine the properties to confirm the asymmetric range.
Create an uncertain parameter, specifying variability with *Percentage*, but force the *Mode* to be *Range*.

\[ c = \text{ureal('c',-5,'Percentage',[-20 30])}; \]
\[ \text{get(c)} \]

\[ \begin{align*}
\text{Name: } & \text{'c'} \\
\text{NominalValue: } & -5 \\
\text{Mode: } & \text{'Percentage'} \\
\text{Range: } & \text{[-6 -3.5000]} \\
\text{PlusMinus: } & \text{[-1 1.5000]} \\
\text{Percentage: } & \text{[-20 30]} \\
\text{AutoSimplify: } & \text{'basic'}
\end{align*} \]

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

\[ d = \text{ureal('d',-1,'Mode','Range','Percentage',[-40 60])}; \]
\[ \text{get(d)} \]

\[ \begin{align*}
\text{Name: } & \text{'d'} \\
\text{NominalValue: } & -1 \\
\text{Mode: } & \text{'Range'} \\
\text{Range: } & \text{[-1.4000 -0.4000]} \\
\text{PlusMinus: } & \text{[-0.4000 0.6000]} \\
\text{Percentage: } & \text{[-40 60]} \\
\text{AutoSimplify: } & \text{'basic'}
\end{align*} \]

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

\[ e = \text{ureal('e',10,'PlusMinus',[-23], 'Mode', 'Percentage', 'AutoSimplify','Full')}; \]
\[ \text{get(e)} \]

\[ e = \]

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

\[ \begin{align*}
\text{Name: } & \text{'e'} \\
\text{NominalValue: } & 10 \\
\text{Mode: } & \text{'Percentage'} \\
\text{Range: } & \text{[-13 33]} \\
\text{PlusMinus: } & \text{[-23 23]} \\
\text{Percentage: } & \text{[-230 230]}
\end{align*} \]
AutoSimplify: 'full'

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.

```matlab
f = ureal('f',3,'PlusMinus',[-2 1],'Percentage',40)
g = ureal('g',2,'PlusMinus',[-2 1],'Mode','Range','Percentage',40)
g.Mode
```

```matlab
f =

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

g =

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

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).

```matlab
h = ureal('h',3);
hsample = usample(h,1000);
hist(reshape(hsample,[1000 1]),20);
```
Create Uncertain Real Parameters

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

```matlab
h.Range = [2 6];
hsample = usample(h,1000);
hist(reshape(hsample,[1000 1]),40);
```
Note that the distribution is skewed. However, the number of samples less than the nominal value and the number of samples greater than the nominal value are equal (on average). Verify this.

\[ \text{length(find(hsample(:) < h.NominalValue))} \]

\[ \text{ans =} \]

\[ 482 \]

\[ \text{length(find(hsample(:) > h.NominalValue))} \]
ans =

518

The distribution used in usample is uniform in the normalized description of the uncertain real parameter. See “Decomposing Uncertain Objects” to learn more about the normalized description.

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.
Uncertain LTI Dynamics Elements

Uncertain linear, time-invariant objects, `ultidyn`, are used to represent unknown linear, time-invariant dynamic objects, whose only known attributes are bounds on their frequency response. 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 $\Delta$ is an `ultidyn` element, and if $\gamma$ 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'`, $\sigma[\Delta(\omega)] \leq \gamma$ for all frequencies. When `Type` is `'GainBounded'`, the default value for `Bound` (i.e., $\gamma$) is 1. The NominalValue of $\Delta$ is always the 0-matrix.

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

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

<table>
<thead>
<tr>
<th>Properties</th>
<th>Meaning</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Internal Name</td>
<td>char</td>
</tr>
<tr>
<td>NominalValue</td>
<td>Nominal value of element</td>
<td>See above</td>
</tr>
<tr>
<td>Type</td>
<td><code>'GainBounded'</code></td>
<td><code>'PositiveReal'</code></td>
</tr>
<tr>
<td>Bound</td>
<td>Norm bound or minimum real</td>
<td>scalar double</td>
</tr>
<tr>
<td>SampleStateDimen</td>
<td>State-space dimension of random samples of this uncertain element</td>
<td>scalar double</td>
</tr>
</tbody>
</table>
Uncertain LTI Dynamics Elements

<table>
<thead>
<tr>
<th>Properties</th>
<th>Meaning</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleMaxFrequency</td>
<td>Maximum natural frequency for random sampling</td>
<td>scalar double</td>
</tr>
<tr>
<td>AutoSimplify</td>
<td>'off'</td>
<td>{'basic'}</td>
</tr>
</tbody>
</table>

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.

You can create a 2-by-3 gain-bounded uncertain linear dynamics element. Verify its size, and check the properties.

```matlab
f = ultidyn('f',[2 3]);
size(f)
an = [2 3]
get(f)
    Name: 'f'
    NominalValue: [2x3 double]
    Type: 'GainBounded'
    Bound: 1
    SampleStateDim: 1
    AutoSimplify: 'basic'
```

**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. To see this, create a `ultidyn`, and view the sample time of several random samples of the element.

```matlab
h = ultidyn('h',[1 1]);
get(usample(h), 'Ts')
an = 0
get(usample(h), 'Ts')
an = 0
get(usample(h), 'Ts')
an = 0
```
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. This is explained and demonstrated in “Interpreting Uncertainty in Discrete Time” on page 1-41.
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.

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

nyquist(usample(g,30))
xlim([-2 10])
ylim([-6 6]);
```
Uncertain Complex Parameters and Matrices

Uncertain Complex Parameters

The `ucomplex` element is the Control Design Block 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

<table>
<thead>
<tr>
<th>Properties</th>
<th>Meaning</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Internal Name</td>
<td>char</td>
</tr>
<tr>
<td>NominalValue</td>
<td>Nominal value of element</td>
<td>double</td>
</tr>
<tr>
<td>Mode</td>
<td>'Range'</td>
<td>'Percentage'</td>
</tr>
<tr>
<td>Radius</td>
<td>Radius of disk</td>
<td>double</td>
</tr>
<tr>
<td>Percentage</td>
<td>Additive variation (percent of Radius)</td>
<td>double</td>
</tr>
<tr>
<td>AutoSimplify</td>
<td>'off'</td>
<td>{'basic'}</td>
</tr>
</tbody>
</table>

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.

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

```matlab
a =

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

```matlab
get(a)
```

```
  Name: 'a'
  NominalValue: 2.0000 - 1.0000i
  Mode: 'Radius'
  Radius: 1
  Percentage: 44.7214
  AutoSimplify: 'basic'
```
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 $2-j$.

```matlab
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 \( \Delta \) is any complex matrix with \( \sigma(\Delta) \leq 1 \). All properties of a `ucomplexm` are can be accessed with `get` and `set`. The properties are

<table>
<thead>
<tr>
<th>Properties</th>
<th>Meaning</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Internal Name</td>
<td>char</td>
</tr>
<tr>
<td>NominalValue</td>
<td>Nominal value of element</td>
<td>double</td>
</tr>
<tr>
<td>WL</td>
<td>Left weight</td>
<td>double</td>
</tr>
<tr>
<td>WR</td>
<td>Right weight</td>
<td>double</td>
</tr>
<tr>
<td>AutoSimplify</td>
<td>'off'</td>
<td>{'basic'}</td>
</tr>
</tbody>
</table>

The simplest construction requires only a name and nominal value. The default left and right weight matrices are identity.

You can create a 4-by-3 `ucomplexm` element, and view its properties.

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

Uncertain Complex Matrix: Name m, 4x3

```
get(m)
```

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

```
m.NominalValue
ans =
1    2    3
4    5    6
7    8    9
10   11   12
```

```
m.WL
ans =
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 generally equal, indicative of the default (identity) weighting matrices that are in place.
abs(usample(m)--m.NominalValue)
ans =
   0.2948   0.1001   0.2867
   0.3028   0.2384   0.2508
   0.3376   0.1260   0.2506
   0.2200   0.3472   0.1657

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, 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, which is
completely expected by choice of the diagonal weighting matrices.

abs(usample(m)--m.NominalValue)
ans =
   0.0091   0.0860   0.2753
   0.0057   0.1717   0.6413
   0.0304   0.2756   1.4012
   0.0527   0.4099   1.8335
Systems with Unmodeled Dynamics

The unstructured uncertain dynamic system Control Design Block, 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., robuststab) do not handle these types of uncertain elements. As such, these elements do not provide a significant amount of usability, and their role in the user's guide is small.

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

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

```matlab
m =

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

```matlab
get(m)
```

```matlab
NominalValue: [2x3 ss]
AutoSimplify: 'basic'
Ts: 0
TimeUnit: 'seconds'
InputName: {3x1 cell}
InputUnit: {3x1 cell}
InputGroup: [1x1 struct]
OutputName: {2x1 cell}
OutputUnit: {2x1 cell}
OutputGroup: [1x1 struct]
Name: 'm'
Notes: {}
UserData: []
```
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.

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 Uncertain Matrices from Uncertain Elements

You can create 2 uncertain real parameters, and then a 3-by-2 uncertain matrix using these uncertain elements.

```matlab
a = ureal('a',3);
b = ureal('b',10,'pe',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
Access Properties of a umat

Use `get` to view the accessible properties of a `umat`.

```matlab
get(M)
NominalValue: [3x2 double]
Uncertainty: [1x1 elementlist]
```

The `NominalValue` is a `double`, obtained by replacing all uncertain elements with their nominal values.

```matlab
M.NominalValue
ans =
    -3.0000    0.1000
    10.0000    3.1000
    1.0000    3.0000
```

The `Uncertainty` property is a structure, whose fields are the uncertain elements (Control Design Blocks) of `M`.

```matlab
M.Uncertainty
ans =
    a: [1x1 ureal]
    b: [1x1 ureal]
```

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

Direct access to the elements is facilitated through `Uncertainty`. Check the `Range` of the uncertain element named 'a' within `M`, then change it.

```matlab
M.Uncertainty.a.Range
ans =
    2     4
M.Uncertainty.a.Range = [2.5 5];
M
UMAT: 3 Rows, 2 Columns
    a: real, nominal = 3, variability = [-0.5 2], 2 occurrences
    b: real, nominal = 10, variability = [-20 20]%, 3 occurrences
```
The change to the uncertain real parameter \( a \) only took place within \( M \). Verify that the variable \( a \) in the workspace is no longer the same as the variable \( a \) within \( M \).

```matlab
isequal(M.Uncertainty.a,a)
ans =
   0
```

Note that combining elements which have a common internal name, but different properties leads to an error. For instance, subtracting the two elements gives an error, not 0.

```matlab
M.Uncertainty.a - a
??? Error using ==> ndlft.lftmask

Elements named 'a' have different properties.
Row and Column Referencing

Standard Row/Column referencing is allowed. Note, however, that single-indexing is only allowed if the umat is a column or a row.

Reconstruct M (if need be), and make a 2-by-2 selection from M

```matlab
a = ureal('a',3);
b = ureal('b',10,'pe',20);
M = [-a 1/b; b a+1/b; 1 3];
M.Uncertainty.a.Range = [2.5 5];
M(2:3,:)
```

```
ans =

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 "ans.NominalValue" to see the nominal value, "get(ans)" to see all properties, and "ans.Uncertainty" to interact with the uncertain elements.

Make a single column selection from M, and use single-index references to access elements of it.

```matlab
h = M([2 1 2 3],2)
```

```
h =

Uncertain matrix with 4 rows and 1 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]%, 1 occurrences
```

Type "h.NominalValue" to see the nominal value, "get(h)" to see all properties, and "h.Uncertainty" to interact with the uncertain elements.

```matlab
h(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
```
Finally, make the (3,2) entry of M uncertain.

\[
M(3,2) = \text{ureal}('c',3, '\text{perc}', 40)
\]

\[
M =
\]

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.Uncertainty" to interact with the uncertain elements.
Matrix Operation on umat Objects

Many matrix operations are allowed, such as matrix-multiply, transpose, and inverse. Combinations of certain (i.e., not uncertain) matrices and uncertain matrices are allowed.

Premultiply $M$ by a 1-by-3 constant matrix, resulting in a 1-by-2 umat.

$$M1 = [2 \ 3 \ 1] * M$$

$M1 =$

Uncertain matrix with 1 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
  c: Uncertain real, nominal = 3, variability = [-40 40]%, 1 occurrences

Type "M1.NominalValue" to see the nominal value, "get(M1)" to see all properties, and "M1.Uncertainty" to interact with the uncertain elements.

Verify that the 1st entry of $M1$ is $-2a + 3b + 1$. Direct subtraction yields a umat without any dependence on uncertain elements. Simplifying the class shows that the result has no uncertain elements, as expected.

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

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.Uncertainty" to interact with the uncertain elements.

simplify(d,'class');

ans =

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

Transpose $M$, form a product, an inverse, and sample the uncertain result. As expected, the result is the 2-by-2 identity matrix.

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

ans(:,1) =
\[
\begin{bmatrix}
1.0000 & -0.0000 \\
-0.0000 & 1.0000 \\
\end{bmatrix}
\]

\text{ans(:,:,2) =}
\[
\begin{bmatrix}
1.0000 & 0.0000 \\
-0.0000 & 1.0000 \\
\end{bmatrix}
\]

\text{ans(:,:,3) =}
\[
\begin{bmatrix}
1.0000 & 0.0000 \\
-0.0000 & 1.0000 \\
\end{bmatrix}
\]
Evaluate Uncertain Elements by Substitution

Uncertain elements can be substituted for using `usubs`. For more information, see “Substitution by usubs” on page 1-57. Here, we illustrate a few special cases.

Substitute all instances of the uncertain real parameter named `a` with the number 4. This results in a `umat`, with dependence on the uncertain real parameters `b` and `c`.

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

```matlab
M2 =

Uncertain matrix with 3 rows and 2 columns.
The uncertainty consists of the following blocks:
  b: Uncertain real, nominal = 10, variability = [-20,20]%, 2 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.Uncertainty" to interact with the uncertain elements.

Similarly, we can substitute all instances of the uncertain real parameter named `b` with `M.Uncertainty.a`, resulting in a `umat` with dependence on the uncertain real parameters `a` and `c`.

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

```matlab
M3 =

Uncertain matrix with 3 rows and 2 columns.
The uncertainty consists of the following blocks:
  a: Uncertain real, nominal = 3, variability = [-0.5,2], 4 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.Uncertainty" to interact with the uncertain elements.

Nominal and/or random instances can easily be specified.

```matlab
M4 = usubs(M,'a','nominal','b','random')
```

```matlab
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
```
The command `usample` also generates multiple random instances of a `umat` (and `uss` and `ufrd`). See “Generate Samples of Uncertain Systems” on page 1-53 for details.

**Lifting a double matrix to a umat**

A not-uncertain matrix may be interpreted as an uncertain matrix that has no dependence on uncertain elements. Use the `umat` command to *lift* a double to the `umat` class.

\[
\text{Md} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}; \\
M = \text{umat}(\text{Md})
\]

\[
M = \text{Uncertain matrix with 2 rows, 3 columns, and no uncertain blocks.}
\]

Type "M.NominalValue" to see the nominal value, "get(M)" to see all properties, and "M.Uncertainty" to interact with the uncertain elements.

High dimensional double matrices can also be lifted. Note from the display that once the matrix is interpreted as a `umat`, the third dimension and beyond are interpreted as array dimensions. See “Array Management for Uncertain Objects” on page 1-60 for more information about how multidimensional arrays of uncertain objects are handled.

\[
\text{Md} = \text{randn}(4,5,6); \\
M = \text{umat}(\text{Md})
\]

\[
M = \text{umat}(\text{Md})
\]

\[
6x1 \text{ array of uncertain matrices with 4 rows, 5 columns, and no uncertain blocks.}
\]

Type "M.NominalValue" to see the nominal value, "get(M)" to see all properties, and "M.Uncertainty" to interact with the uncertain elements.

\[
\text{Md} = \text{randn}(4,5,6,7); \\
M = \text{umat}(\text{Md})
\]

\[
M = \text{umat}(\text{Md})
\]

\[
6x7 \text{ array of uncertain matrices with 4 rows, 5 columns, and no uncertain blocks.}
\]

Type "M.NominalValue" to see the nominal value, "get(M)" to see all properties, and "M.Uncertainty" to interact with the uncertain elements.
Uncertain State-Space Models (uss)

Uncertain systems (uss) are linear systems with uncertain state-space matrices and/or uncertain linear dynamics. Like their certain (i.e., not uncertain) counterpart, the ss object, they are often built from state-space matrices using the ss command. In the case where some of the state-space matrices are uncertain elements (uncertain Control Design Blocks), the result will be a uncertain state-space (uss) object.

Combining uncertain systems with uncertain systems (with the feedback command, for example) usually leads to an uncertain system. Not-uncertain systems can be combined with uncertain systems. Usually the result is an uncertain system.

The nominal value of an uncertain system is a ss object, which is familiar to Control System Toolbox software users.

Properties of uss Objects

Create a uss object and view the properties.

```matlab
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); % Create uss model
get(sys)
```

\[
\begin{align*}
a & : [2x2 \text{ umat}] \\
b & : [2x1 \text{ umat}] \\
c & : [2x2 \text{ umat}] \\
d & : [2x1 \text{ double}] \\
e & : [
\\
\text{StateName} & : [2x1 \text{ cell}] \\
\text{StateUnit} & : [2x1 \text{ cell}] \\
\text{NominalValue} & : [2x1 \text{ ss}] \\
\text{Uncertainty} & : [1x1 \text{ struct}] \\
\text{InternalDelay} & : [0x1 \text{ double}] \\
\text{InputDelay} & : 0
\end{align*}
\]
OutputDelay: [2x1 double]
  Ts: 0
  TimeUnit: 'seconds'
  InputName: {''
  InputUnit: {''
  InputGroup: [1x1 struct]
  OutputName: {2x1 cell}
  OutputUnit: {2x1 cell}
  OutputGroup: [1x1 cell]
  Name: '
  Notes: {}
  UserData: []
  SamplingGrid: [1x1 struct]

The properties a, b, c, d, and StateName behave in exactly the same manner as in Control System Toolbox ss objects. The properties InputName, OutputName, InputGroup and OutputGroup behave in exactly the same manner as all of the Control System Toolbox objects (ss, zpk, tf, and frd).

The NominalValue is a Control System Toolbox ss object, and hence all methods for ss objects are available. For instance, compute the poles and step response of the nominal system.

```
pole(sys.NominalValue)

ans =
  -10
  -10
```

```
step(sys.NominalValue)
```
Just as with the 
emat class, the 
uncertainty property is a structure containing the 
uncertain elements. Direct access to the elements is facilitated through 
uncertainty. 
Check the Range of the uncertain element named 
'p2' within 
sys, then change its left 
endpoint.

sys.Uncertainty.p2.range

ans =

2.5000    4.2000

sys.Uncertainty.p2.range(1) = 2;
Create Uncertain State-Space Model

Uncertain systems (class `uss`) are built from certain and/or uncertain state-space matrices, usually using the `ss` command.

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.

You can create three uncertain real parameters. Then create 3 uncertain matrices A, B and C, and one `double` matrix D.

```matlab
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];
```

Pack the 4 matrices together using the `ss` command. This results in a continuous-time 2-output, 1-input, 2-state uncertain system.

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

```
USS: 2 States, 2 Outputs, 1 Input, Continuous System
  p1: real, nominal = 10, variability = [-50 50]%, 2 occurrences
  p2: real, nominal = 3, variability = [-0.5 1.2], 2 occurrences
  p3: real, nominal = 0, variability = [-1 1], 2 occurrences
```
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
```

```
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`. 
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.

```matlab
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.

```matlab
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.

```matlab
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`.

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

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

```matlab
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.

```matlab
subplot(2,1,1); stepplot(P,5)
textplot(P,5) subplot(2,1,2); stepplot(CLP,5)
textplot(CLP,5)
```
Building Uncertain Models

Step Response

Amplitude vs. Time (seconds)

Step Response

Amplitude vs. Time (seconds)
Interpreting Uncertainty in Discrete Time

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

For example, create a scalar \texttt{ultidyn} object. Then, create two 1-input, 1-output \texttt{uss} objects using the \texttt{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.

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
[sys1s,d1v] = usample(sys1);
[sys2s,d2v] = usample(sys2);
\end{verbatim}

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

\begin{verbatim}
d1v.delta.Ts
ans =
0

d2v.delta.Ts
ans =
0.4200
\end{verbatim}

Finally, in the case of a discrete-time \texttt{uss} object, it is not the case that \texttt{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.
**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.

```matlab
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 1-60 for more information about how arrays of uncertain objects are handled.
Create Uncertain Frequency Response Data Models

Uncertain frequency responses (\textit{ufrd}) arise naturally when computing the frequency response of an uncertain state-space (\textit{uss}). They also arise when frequency response data (in an \textit{frd} model object) is combined (added, multiplied, concatenated, etc.) to an uncertain matrix (\textit{umat}).

The most common manner in which a \textit{ufrd} arises is taking the frequency response of a \textit{uss}. To do this, use the \textit{ufrd} command.

Construct an uncertain state-space model \textit{sys}.

\begin{verbatim}
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)
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

Compute the uncertain frequency response of the uncertain system. Use the \textit{ufrd} command, along with a frequency grid containing 100 points. The result is an uncertain frequency response data object, referred to as a \textit{ufrd} model.

\begin{verbatim}
sysg = ufrd(sys,logspace(-2,2,100))
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
Properties of ufrd Objects

View the properties with the `get` command.

```plaintext
get(sysg)
    Frequency: [100x1 double]
    FrequencyUnit: 'rad/TimeUnit'
    ResponseData: [4-D 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]
    Name: ''
    Notes: {}
    UserData: []
```

The properties `ResponseData` and `Frequency` behave in exactly the same manner as Control System Toolbox `frd` objects, except that `ResponseData` is a `umat`. The properties `InputName`, `OutputName`, `InputGroup` and `OutputGroup` behave in exactly the same manner as all of the Control System Toolbox objects (`ss`, `zpk`, `tf` and `frd`).

The `NominalValue` is a Control System Toolbox `frd` object, and hence all methods for `frd` objects are available. For instance, plot the Bode response of the nominal system.

```plaintext
bode(sysg.nom)
```
Just as with the `umat` and `uss` classes, the `Uncertainty` property is a structure containing the uncertain elements. Direct access to the elements is facilitated through `Uncertainty`. Change the nominal value of the uncertain element named `'p1'` within `sysg` to 14, and replot the Bode plot of the (new) nominal system.

```matlab
sysg.unc.p1.nom = 14
```

**UFRD: 2 Outputs, 1 Input, Continuous System, 100 Frequency points**
- `p1`: real, nominal = 14, variability = [-50 50]%, 2 occurrences
- `p2`: real, nominal = 3, variability = [-0.5 1.2], 2 occurrences
- `p3`: real, nominal = 0, variability = [-1 1], 2 occurrences
Interpreting Uncertainty in Discrete Time

See “Interpreting Uncertainty in Discrete Time” on page 1-41. The issues are identical.
Lifting an frd to a ufrd

A not-uncertain frequency response object may be interpreted as an uncertain frequency response object that has no dependence on uncertain elements. Use the ufrd command to “lift” an frd object to the ufrd class.

```matlab
sys = rss(3,2,1);
sysg = frd(sys,logspace(-2,2,100));
usysg = ufrd(sysg)
UFRD: 2 Outputs, 1 Input, Continuous System, 100 Frequency points
```

Arrays of frd objects can also be lifted. See “Array Management for Uncertain Objects” on page 1-60 for more information about how arrays of uncertain objects are handled.
Basic Control System Toolbox and MATLAB Interconnections

This list has all of the basic system interconnection functions defined in Control System Toolbox software or in MATLAB.

• append
• blkdiag
• series
• parallel
• feedback
• lft
• stack

These functions work with uncertain objects as well. Uncertain objects may be combined with certain objects, resulting in an uncertain object.
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

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

However, a “natural” construction, formed by

```matlab
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 =
\begin{bmatrix}
\begin{array}{cccc}
x1 & x2 & x3 & x4 \\
x1 & -1 & 0 & 0 & 0 \\
x2 & 0 & -1 & 0 & 0 \\
x3 & 0 & 0 & -1 & 0 \\
x4 & 0 & 0 & 0 & -1
\end{array}
\end{bmatrix}

b =
\begin{bmatrix}
\begin{array}{cc}
u1 & u2 \\
x1 & 2 & 0 \\
x2 & 0 & 2 \\
x3 & 2 & 0 \\
x4 & 0 & 2
\end{array}
\end{bmatrix}

c =
\begin{bmatrix}
\begin{array}{cccc}
x1 & x2 & x3 & x4 \\
y1 & 1 & 2 & 0 & 0 \\
y2 & 0 & 0 & 1.5 & 3
\end{array}
\end{bmatrix}

d =
\begin{bmatrix}
\begin{array}{cc}
u1 & u2 \\
y1 & 0 & 0 \\
y2 & 0 & 0
\end{array}
\end{bmatrix}
```
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 one of the strings '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.

```matlab
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`.

```matlab
a.AutoSimplify = 'off';
m2 = [a+4 6*a]
```
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.

```matlab
a.AutoSimplify = 'basic';
m3 = [a*(a+4) 6*a*a]
```

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`.

```matlab
a.AutoSimplify = 'full';
m4 = [a*(a+4) 6*a*a]
```

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.

```matlab
usubs(m3,'a',0)
an =
   0   0
usubs(m4,'a',0)
an =
   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.

```matlab
usubs(m3,'a',1)
an =
    5   6
```
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'.

```matlab
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`.

```matlab
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.
Generate Samples of Uncertain Systems

The command `usample` is used to randomly sample an uncertain object, giving a not-uncertain instance of the uncertain object.

**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`.

```matlab
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`.

```matlab
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.

```matlab
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.

```matlab
size(usample(M,30))
ans =
```
See “Create Arrays with usample” on page 1-66 for more information on sampling uncertain objects.

**Sampling ultidyn Elements**

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.

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

```matlab
del = ultidyn('del',[1 1],'Bound',3);
del.SampleStateDimension
```

```
ans =
     1
```

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.

```matlab
delS = usample(del,30);
size(delS)
```

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

Plot the Nyquist plot of these samples and add a disk of radius 3. Note that the gain bound is satisfied and that the Nyquist plots are all circles, indicative of 1st order systems.

```matlab
nyquist(delS)
hold on;
theta = linspace(-pi,pi);
plot(del.Bound*exp(sqrt(-1)*theta),'r');
hold off;
```
Change `SampleStateDimension` to 4, and repeat entire procedure. The Nyquist plots satisfy the gain bound and as expected are more complex than the circles found in the 1st-order sampling.

```matlab
del.SampleStateDimension = 4;
delS = usample(del,30);
nyquist(delS)
hold on;
theta = linspace(-pi,pi);
plot(del.Bound*exp(sqrt(-1)*theta),'r');
hold off;
```
Nyquist Diagram

Imaginary Axis

Real Axis

1 Building Uncertain Models
Substitution by usubs

If an uncertain object (umat, uss, ufrd) has many uncertain parameters, it is often necessary to freeze some, but not all, of the uncertain parameters to specific values. 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. You 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.Uncertainty" to interact with the uncertain elements.

Use usubs to substitute the uncertain element named delta in A with the value 2.3, leaving all other uncertain elements intact. Note 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.Uncertainty" to interact with the uncertain elements.

To set multiple elements, list individually, or group the values in a data structure. For instance, the following are the same.
B1 = usubs(A,'\text{delta}',2.3,'\text{eta}',A.Uncertainty.rho);
S.delta = 2.3;
S.eta = A.Uncertainty.rho;
B2 = usubs(A,S);

In each case, \text{delta} is replaced by 2.3, and \text{eta} is replaced by A.Uncertainty.rho.

Any superfluous substitution requests are ignored. Hence, the following returns an uncertain model that is the same as A.

B4 = usubs(A,'\text{fred}',5);

\textbf{Specifying the Substitution with Structures}

An alternative syntax for \texttt{usubs} is to specify the substituted values in a structure, whose fieldnames are the names of the elements being substituted with values.

Create a structure \texttt{NV} with 2 fields, \texttt{delta} and \texttt{eta}. Set the values of these fields to be the desired substituted values. Then perform the substitution with \texttt{usubs}.

\begin{verbatim}
NV.delta = 2.3;
NV.eta = A.Uncertainty.rho;
B6 = usubs(A,NV);
\end{verbatim}

Here, B6 is the same as B1 and B2 above. Again, any superfluous fields are ignored. Therefore, adding an additional field \texttt{gamma} to \texttt{NV}, and substituting does not alter the result.

\begin{verbatim}
NV.gama = 0;
B7 = usubs(A,NV);
\end{verbatim}

Here, B7 is the same as B6.

The commands \texttt{wcgain}, \texttt{robuststab} and \texttt{usample} all return substitutable values in this structure format. More discussion can be found in “Create Arrays with \texttt{usubs}” on page 1-68.

\textbf{Nominal and Random Values}

If the replacement value is the (partial and case-independent) string ‘Nominal’, then the listed element are replaced with their nominal values. Therefore

B8 = usubs(A,fieldnames(A.Uncertainty),'nom')
B8 =

\[
\begin{align*}
11.0000 & \quad 0.3333 \\
6.0000 & \quad 11.0000
\end{align*}
\]

B9 = A.NominalValue

B9 =

\[
\begin{align*}
11.0000 & \quad 0.3333 \\
6.0000 & \quad 11.0000
\end{align*}
\]

are the same. It is possible to only set some of the elements to NominalValues, and would be the typical use of usubs with the 'nominal' argument.

Within A, set eta to its nominal value, delta to a random value (within its range) and rho to a specific value, say 6.5

B10 = usubs(A,'eta','nom','delta','rand','rho',6.5)

B10 =

\[
\begin{align*}
10.5183 & \quad 0.2531 \\
13.5000 & \quad 15.6100
\end{align*}
\]

Unfortunately, the 'nominal' and 'Random' specifiers may not be used in the structure format. However, explicitly setting a field of the structure to an element's nominal value, and then following (or preceding) the call to usubs with a call to usample (to generate the random samples) is acceptable, and achieves the same effect.
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 $\text{size}(M)$ equals $[4 \ 5 \ 3 \ 6 \ 7]$, then (for example) the size of $M([4 \ 2],[1 \ 2 \ 4])$ is $[2 \ 3 \ 6 \ 7]$. If $\text{size}(M,1)==1$ or $\text{size}(M,2)==1$, 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, $\text{size}(G,1)$ equals $\text{length}(Yidx)$, $\text{size}(G,2)$ equals $\text{length}(Uidx)$, $\text{size}(G,3)$ equals $\text{length}(idx1)$, $\text{size}(G,4)$ equals $\text{length}(idx2)$, and $\text{size}(G,K+2)$ equals $\text{length}(idxK)$. 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 $\text{size}(M)$ 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 $\text{size}(G)$ 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?”).
Create Arrays with stack and cat Functions

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

\begin{verbatim}
a = ureal('a',4);
b = ureal('b',2);
M = stack(1,[a b 1],[-a -b 4+a],[4 5 6],[a 0 0])
\end{verbatim}

\texttt{UMAT: 1 Rows, 3 Columns [array, 4 x 1]}
\begin{itemize}
  \item \texttt{a}: real, nominal = 4, variability = \([-1 \quad 1]\), 1 occurrence
  \item \texttt{b}: real, nominal = 2, variability = \([-1 \quad 1]\), 1 occurrence
\end{itemize}

\begin{verbatim}
size(M)
\end{verbatim}
\begin{verbatim}
ans =
   1   3   4
\end{verbatim}

\begin{verbatim}
arraysize(M)
\end{verbatim}
\begin{verbatim}
ans =
   4   1
\end{verbatim}

Check that result is valid. Use referencing to access parts of the \([4\text{-by-1}]\) \texttt{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 \texttt{a}, respectively.

\begin{verbatim}
simplify(M(:,:,1) - [a b 1])
\end{verbatim}
\begin{verbatim}
simplify(M(:,:,2) - [-a -b 4+a])
\end{verbatim}
\begin{verbatim}
simplify(M(:,:,3) - [4 5 6])
\end{verbatim}
\begin{verbatim}
simplify(M(:,:,4) - [a 0 0])
\end{verbatim}
\begin{verbatim}
simplify(M(1,2,3)) \% should be 5
\end{verbatim}
\begin{verbatim}
simplify(M(1,3,2)-4)
\end{verbatim}

\texttt{Uncertain Real Parameter: Name a, NominalValue 4, variability = \([-1 \quad 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 \texttt{umat}.

\[
N = \text{randn}(1,3,4);
M2 = \text{stack}(2,M,N);
\]

\[
\text{size}(M2)
\]

\[
\begin{array}{cccc}
1 & 3 & 4 & 2 \\
\end{array}
\]

\[
\text{arraysize}(M2)
\]

\[
\begin{array}{cc}
4 & 2 \\
\end{array}
\]

As expected, both \( M \) and \( N \) can be recovered from \( M2 \).

\[
d1 = \text{simplify}(M2(:,:,1)-M);
d2 = \text{simplify}(M2(:,:,2)-N);
\]

\[
\left[ \text{max(abs(d1(:))) max(abs(d2(:)))} \right]
\]

\[
\begin{array}{c}
0 \\
0 \\
\end{array}
\]

It is also possible to stack \( M \) and \( N \) along the 1st array dimension, creating a 1-by-3-by-8-by-1 \texttt{umat}.

\[
M3 = \text{stack}(1,M,N);
\]

\[
\text{size}(M3)
\]

\[
\begin{array}{ccc}
1 & 3 & 8 \\
\end{array}
\]

\[
\text{arraysize}(M3)
\]

\[
\begin{array}{cc}
8 & 1 \\
\end{array}
\]

As expected, both \( M \) and \( N \) can be recovered from \( M3 \).

\[
d3 = \text{simplify}(M3(:,:,1:4)-M);
d4 = \text{simplify}(M3(:,:,5:8)-N);
\]

\[
\left[ \text{max(abs(d3(:))) max(abs(d4(:)))} \right]
\]

\[
\begin{array}{c}
0 \\
0 \\
\end{array}
\]
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

```matlab
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,

```matlab
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 \( C = \text{fcn}(A, B) \) is equivalent to looping on \( k1, k2, \ldots \), setting

\[
C(:,:,k1,k2,\ldots) = \text{fcn}(A(:,:,k1,k2,\ldots), B(:,:,k1,k2,\ldots))
\]

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 \( l_A = l_B \), 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: \( n_k = m_k \), or \( n_k = 1 \), or \( m_k = 1 \). 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.
Create Arrays with usample

An extremely common manner in which to generate an array is to sample (in some of the uncertain elements) an uncertain object. Using the ureal objects a and b from above, create a 2-by-3 umat.

\[
M = \begin{bmatrix}
a & b \\
b & a/b \\
1-b & 1+a*b
\end{bmatrix}
\]

UMAT: 3 Rows, 2 Columns
  a: real, nominal = 4, variability = [-1 1], 3 occurrences
  b: real, nominal = 2, variability = [-1 1], 6 occurrences

\[
\text{size}(M)
\]
\[
\text{ans} =
\begin{bmatrix}
3 & 2
\end{bmatrix}
\]

Sample (at 20 random points within its range) the uncertain real parameter b in the matrix M. This results in a 3-by-2-by-20 umat, with only one uncertain element, a. The uncertain element b of M has been “sampled out”, leaving a new array dimension in its place.

\[
[M_s, \text{bvalues}] = \text{usample}(M, 'b', 20);
\]

\[M_s\]
UMAT: 3 Rows, 2 Columns [array, 20 x 1]
  a: real, nominal = 4, variability = [-1 1], 2 occurrences

\[
\text{size}(M_s)
\]
\[
\text{ans} =
\begin{bmatrix}
3 & 2 & 20
\end{bmatrix}
\]

Continue sampling (at 15 random points within its range) the uncertain real parameter a in the matrix Ms. This results in a 3-by-2-by-20-by-15 double.

\[
[M_{ss}, \text{avvalues}] = \text{usample}(M_s, 'a', 15);
\]

\[M_{ss}\]
size(M_{ss})
\[
\text{ans} =
\begin{bmatrix}
3 & 2 & 20 & 15
\end{bmatrix}
\]

\[
\text{class}(M_{ss})
\]
\[
\text{ans} =
\text{double}
\]

The above 2-step sequence can be performed in 1 step.

\[
[M_{ss}, \text{values}] = \text{usample}(M, 'b', 20, 'a', 15);
\]

\[M_{ss}\]
\[
\text{class}(M_{ss})
\]
\[
\text{ans} =
\text{double}
\]
In this case, `values` is a 20-by-15 struct array, with 2 fields `b` and `a`, whose values are the values used in the random sampling. It follows that `usubs(M,values)` is the same as `Mss`.

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

```matlab
[M,values] = usample(M,{'a' 'b'},800);
size(M)
ans =
  3   2  800
size(values)
ans =
  800   1
```
Create Arrays with usubs

Suppose Values is a struct array, with the following properties: the dimensions of Value match the array dimensions of M, as described in “Create Arrays with usample” on page 1-66; the field names of Values are some (or all) of the names of the uncertain elements of M; and the dimensions of the contents of the fields within Values match the sizes of the uncertain elements within M. Then usubs(M,Values) will substitute the uncertain elements in M with the contents found in the respective fields of Values.

You can create a 3-by-2 uncertain matrix using two uncertain real parameters.

\[
\begin{align*}
a &= \text{ureal}'a',4; \\
b &= \text{ureal}'b',2; \\
M &= [a\ b; b^2 a/b; 1-b 1+a*b];
\end{align*}
\]

Create a 5-by-1 struct array with field name a. Make its values random scalars. Create a 1-by-4 struct array with field name b.

\[
\begin{align*}
&\text{Avalue} = \text{struct}'a',\text{num2cell}.rand(5,1)); \\
&\text{Bvalue} = \text{struct}'b',\text{num2cell}.rand(1,4));
\end{align*}
\]

Substitute the uncertain real parameter a in M with the values in Avalue, yielding ma. Similarly substitute the uncertain real parameter b in M with the values in Avalue, yielding mb.

\[
\begin{align*}
&\text{ma} = \text{usubs}(M,Avalue) \\
&\text{UMAT: 3 Rows, 2 Columns [array, 5 x 1]} \\
&\quad \text{b: real, nominal = 2, variability = [1 1], 6 occurrences} \\
&\text{mb} = \text{usubs}(M,Bvalue) \\
&\text{UMAT: 3 Rows, 2 Columns [array, 1 x 4]} \\
&\quad \text{a: real, nominal = 4, variability = [1 1], 2 occurrences}
\end{align*}
\]

Continue, substituting the uncertain real parameter b in ma with the values in Bvalue, yielding mab. Do the analogous operation for mb, yielding mba. Subtract, and note that the difference is 0, as expected.

\[
\begin{align*}
&\text{mab} = \text{usubs}(ma,Bvalue); \\
&\text{mba} = \text{usubs}(mb,Avalue); \\
&\text{thediff} = \text{mab-mba}; \\
&\text{max(abs(thediff(:)))} \\
&\text{ans =} \\
&\quad 4.4409e-016
\end{align*}
\]
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.

\[
a = \text{ureal}('a',3, 'Range', [2.5 \ 4]); \\
b = \text{ureal}('b',4, 'Percentage', 15); \\
c = \text{ureal}('c',-2, 'Plusminus', [-1 \ .3]); \\
M = [a \ b; b \ c] \\
\]

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 \(c\).

\[
M_{\text{grid1}} = \text{gridureal}(M, 'b', 100) \\
\]

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.

\[
M_{\text{grid3}} = \text{gridureal}(M, 'a', 20, 'b', 12, 'c', 7); \\
\text{size(Mgrid3)} \\
\]

\[
\text{ans} = \\
2 \quad 2 \quad 20 \quad 12 \quad 7
\]
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.

```matlab
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)
an =
 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`.

```matlab
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)
an =
 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(:)))]
an =
 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

Replicating and tiling uncertain state-space systems (uss, and uncertain frequency response data (ufrd) is done 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 object (umat, uss, ufrd) 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, 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, ucomplex and ucomplexm do not have their NominalValue necessarily at zero, and in the case of ureal objects, are not 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.

If $\rho$ 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 $\rho$ satisfying $L \leq \rho \leq R$, it follows that $-1 \leq F(\rho) \leq 1$, 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.

If $E$ is an uncertain gain-bounded, linear, time-invariant dynamic uncertainty, with gain-bound $\beta$, 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 $\beta$, 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 $a = 2|\beta| + 1$.

The normalizing function for an uncertain complex parameter $\xi$, with nominal value $C$ and radius $\gamma$, is

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

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

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

In each case, 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.

**Properties of the Decomposition**

Take an uncertain object $A$, dependent on uncertain real parameters $\rho_1,...,\rho_N$, uncertain complex parameters $\xi_1,...,\xi_K$, uncertain complex matrices $H_1,...,H_B$, uncertain gain-bounded linear, time-invariant dynamics $E_1,...,E_D$, and uncertain positive-real linear, time-invariant dynamics $P_1,...,P_Q$. 
Write $A(\rho, \xi, H, E, P)$ to indicate this dependence. Using `lftdata`, $A$ can be decomposed into two separate pieces: $M$ and $\Delta(\rho, \xi, H, E, P)$ 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`).
- $\Delta$ is always a `umat`, depending on the same uncertain elements as $A$, with ranges, bounds, weights, etc., unaltered.
- The form of $\Delta$ is block diagonal, with elements made up of the normalizing functions acting on the individual uncertain elements:

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

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

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

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)`. See “Advanced Syntax of `lftdata`” on page 1-77 for more information.

**Syntax of `lftdata`**

The decomposition is carried out by the command `lftdata`.

You can create a 2-by-2 `umat` named $A$ using three uncertain real parameters.

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

**UMAT: 2 Rows, 2 Columns**

- $\text{delta: real, nominal = 2, variability = [-1 1], 2 occurrences}$
- $\text{eta: real, nominal = 6, variability = [-1 1], 3 occurrences}$
- $\text{rho: real, nominal = -1, variability = [-1 1], 1 occurrence}$
Note that $A$ depends on two occurrences of $\delta$, three occurrences of $\eta$ and one occurrence of $\rho$.

Decompose $A$ into $M$ and $\Delta$. Note that $A$ is a double, and $\Delta$ has the same uncertainty dependence as $M$.

$$ [M, \Delta] = \text{lftdata}(A); $$

```
class(M)
ans =
double
```

Delta
UMAT: 6 Rows, 6 Columns
  $\delta$: real, nominal = 2, variability = [-1  1], 2 occurrences
  $\eta$: real, nominal = 6, variability = [-1  1], 3 occurrences
  $\rho$: real, nominal = -1, variability = [-1  1], 1 occurrence

Sample $\Delta$ at 5 points. Things to note are:

- It is diagonal.
- The values range between -1 and 1.
- There are three independent values, and duplication of the entries is consistent with the dependence of $\Delta$ and $A$ on the three uncertain real parameters.

```
usample(\Delta, 5)
ans(:, :, 1) =
  -0.7106   0   0   0   0   0
  0  -0.7106   0   0   0   0
  0   0  0.6374   0   0   0
  0   0   0  0.6374   0   0
  0   0   0   0  0.6374   0
  0   0   0   0   0 -0.1258
ans(:, :, 2) =
  -0.5850   0   0   0   0   0
  0  -0.5850   0   0   0   0
  0   0 -0.3021   0   0   0
  0   0   0 -0.3021   0   0
  0   0   0   0 -0.3021   0
  0   0   0   0   0  0.0803
ans(:, :, 3) =
  0.7013   0   0   0   0   0
  0  0.7013   0   0   0   0
  0   0 -0.6749   0   0   0
  0   0   0 -0.6749   0   0
  0   0   0   0 -0.6749   0
  0   0   0   0   0   0
```
Decomposing Uncertain Objects

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & -0.6749 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.3967 \\
\end{bmatrix}
\]

\[
\text{ans(:,4)} = \\
\begin{bmatrix}
0.4262 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.4262 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.0795 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.0795 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.0795 & 0 \\
0 & 0 & 0 & 0 & 0 & -0.9959 \\
\end{bmatrix}
\]

\[
\text{ans(:,5)} = \\
\begin{bmatrix}
-0.8392 & 0 & 0 & 0 & 0 & 0 \\
0 & -0.8392 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.8467 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.8467 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.8467 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.6732 \\
\end{bmatrix}
\]

In fact, verify that the maximum gain of \( \Delta \) is indeed 1

\[
\text{maxnorm} = \text{wcnorm}(\Delta)
\]
\[
\text{maxnorm} = \\
\begin{bmatrix}
\text{LowerBound}: 1.0000 \\
\text{UpperBound}: 1.0004 \\
\end{bmatrix}
\]

Finally, verify that \( \text{lft}(\Delta, M) \) is the same as \( \text{A Subtract} \) (and use the 'full' option in \( \text{simplify} \))

\[
\text{simplify(lft(\Delta,M)-A,'full')}
\]
\[
\text{ans} = \\
\begin{bmatrix}
0 & 0 \\
0 & 0 \\
\end{bmatrix}
\]

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.

\[
\text{[M,Delta,BlkStruct,NormUnc]} = \text{lftdata}(A);
\]

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

Note that the range/bound information about each uncertain element is not included in \texttt{BlkStruct}.

\begin{verbatim}
BlkStruct(1)
ans =
  Name: 'delta'
  Size: [1 1]
  Type: 'ureal'
  Occurrences: 2

BlkStruct(2)
ans =
  Name: 'eta'
  Size: [1 1]
  Type: 'ureal'
  Occurrences: 3

BlkStruct(3)
ans =
  Name: 'rho'
  Size: [1 1]
  Type: 'ureal'
  Occurrences: 1
\end{verbatim}

Together, these mean \( \text{Delta} \) is a block diagonal augmentation of the normalized version of 3 uncertain elements.

The first element is named \texttt{'delta'}. It is 1-by-1; it is of class \texttt{ureal}; and there are 2 copies diagonally augmented.

The second element is named \texttt{'eta'}. It is 1-by-1; it is of class \texttt{ureal}; and there are 3 copies diagonally augmented.

The third element is named \texttt{'rho'}. It is 1-by-1; it is of class \texttt{ureal}; and there is 1 copy.

The 4th output argument 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 \( A \).

\begin{verbatim}
size(NormUnc)
ans =
   6    1

NormUnc{1}
Uncertain Real Parameter: Name deltaNormalized, NominalValue 0,
\end{verbatim}
variability = [-1 1]
isequal(NormUnc{2},NormUnc{1})
ans =
   1
NormUnc{3}
Uncertain Real Parameter: Name etaNormalized, NominalValue 0,
variability = [-1 1]
isequal(NormUnc{4},NormUnc{3})
ans =
   1
isequal(NormUnc{5},NormUnc{3})
ans =
   1
NormUnc{6}
Uncertain Real Parameter: Name rhoNormalized, NominalValue 0,
variability = [-1 1]

Each normalized element has the string 'Normalized' appended to its original name to avoid confusion. By normalized,

- **ureal** objects have nominal value of 0, and range from -1 to 1.
- **ultidyn** objects are norm bounded, 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 power-user has direct access to all of the linear fractional transformation details, and can easily work at the level of the theorems and algorithms that underlie the methods.
Generalized Robustness Analysis

• “Generalized Robustness Analysis” on page 2-2
• “Robust Stability Margin” on page 2-4
• “Robust Performance Margin” on page 2-5
• “Worst-Case Gain Measure” on page 2-6
Generalized Robustness Analysis

The performance of a nominally stable uncertain system model will generally degrade for specific values of its uncertain elements. Moreover, the maximum possible degradation increases as the uncertain elements are allowed to further and further deviate from their nominal values.

The graph below shows the typical tradeoff curve between allowable deviation of uncertain elements from their nominal values and the worst-case degradation in system performance. Here, system performance is characterized by system gain (e.g., peak magnitude on Bode plot). Interpreting the system as the relationship mapping disturbances/commands to errors, small system gains are desirable, and large gains are undesirable.

When all uncertain elements are set to their nominal values (i.e., zero deviation from their nominal values) the input/output gain of the system is its nominal value. In the figure, the nominal system gain is about 0.8. As the uncertainties are allowed to deviate from nominal, the maximum (over the uncertain elements) system gain increases. The
heavy blue line represents the maximum system gain due to uncertainty of various sizes (the horizontal axis) and is called the system performance degradation curve. It is monotonically increasing.

Determining specific attributes of the system performance degradation curve are referred to as robustness computations.

Generally, “robustness computations” refer to determining specific attributes of the system performance degradation curve. The commands robuststab, robustperf and wcgain all compute single scalar attributes of the system performance degradation curve.

Redraw the system performance degradation curve with 3 additional curves: a hyperbola defined by $xy=1$; a vertical line drawn at the uncertainty bound $= 1$; and a vertical line tangent to the asymptotic behavior of the performance degradation curve at large uncertainty bounds. These are used to define three robustness measures, explained next.
Robust Stability Margin

The robust stability margin, StabMarg, is the size of the smallest deviation from nominal of the uncertain elements that leads to system instability.

System instability is equivalent to the system gain becoming arbitrarily large, and hence characterized by the vertical line tangent to the asymptotic behavior of the performance degradation curve.
Robust Performance Margin

The hyperbola is used to define the performance margin. Systems whose performance degradation curve intersects high on the hyperbola curve represent “non-robustly performing systems” in that very small deviations of the uncertain elements from their nominal values can result in very large system gains. Conversely, an intersection low on the hyperbola represent “robustly performing systems.”

The point where the system performance degradation curve crosses the green line is used as a scalar measure of the robustness of a system to uncertainty. The horizontal coordinate of the crossing point is the robust performance margin, PerfMarg.
Worst-Case Gain Measure

The worst-case gain measure is the maximum achievable system gain over all uncertain elements whose normalized size is bounded by 1.

On the graph, this is the vertical coordinate of the performance degradation curve as it crosses the vertical line drawn at the uncertainty bound = 1.

Each measure captures a single scalar attribute of the system performance degradation curve. Mathematically, they are independent quantities, answering subtly different questions. Consequently, for two uncertain systems, sysA and sysB, it is possible that the StabMarg of sysA is larger than the StabMarg of sysB, though the PerfMarg of sysA is smaller than the PerfMarg of sysB. Nevertheless, they are useful metrics for concise description of the robustness of a system (uss or ufrd) due to various uncertain elements.
Introduction to Linear Matrix Inequalities

• “Linear Matrix Inequalities” on page 3-2
• “LMIs and LMI Problems” on page 3-4
• “LMI Applications” on page 3-6
• “Further Mathematical Background” on page 3-10
• “Bibliography” on page 3-11
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 4-20).
- 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 $H_2$, $H_\infty$, and pole placement objectives (h2hinfsyn)
- Functions for synthesizing robust gain-scheduled $H_\infty$ 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 $o(n^5)$. 
LMIs and LMI Problems

A linear matrix inequality (LMI) is any constraint of the form

\[ A(x) := A_0 + x_1 A_1 + \ldots + x_N A_N < 0 \]

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 3-1 since they can be rewritten as \(-A(x) < 0\) and \(A(x) - B(x) < 0\), respectively.

The LMI of Equation 3-1 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 \( \mathbb{R}^N \)
- Finding a solution \( x \) to Equation 3-1, if any, is a convex optimization problem.

Convexity has an important consequence: even though Equation 3-1 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

\[
\begin{aligned}
A_1(x) &< 0 \\
\vdots \\
A_K(x) &< 0
\end{aligned}
\]

is equivalent to

\[ A(x) := \text{diag}(A_1(x), \ldots, A_K(x)) < 0 \]

where \( \text{diag}(A_1(x), \ldots, A_K(x)) \) denotes the block-diagonal matrix with
$A_1(\alpha), \ldots, A_K(\alpha)$ 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 3-1, but rather in the form

$$L(X_1, \ldots, X_n) < R(X_1, \ldots, X_n)$$

where $L(.)$ and $R(.)$ are affine functions of some structured matrix variables $X_1, \ldots, X_n$. A simple example is the Lyapunov inequality

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

where the unknown $X$ is a symmetric matrix. Defining $x_1, \ldots, x_N$ as the independent scalar entries of $X$, this LMI could be rewritten in the form of Equation 3-1. Yet it is more convenient and efficient to describe it in its natural form Equation 3-2, which is the approach taken in the LMI Lab.
LMI Applications

Finding a solution $x$ to the LMI system

$$A(x) < 0$$

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

$$A(x) < 0$$

plays an important role in LMI-based design. Finally, the *generalized eigenvalue minimization problem*:

Minimize $\lambda$ subject to

$$A(x) < \lambda B(x)$$

$$B(x) > 0$$

$$C(x) > 0$$

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, $H^\infty$ control, covariance control, etc. Further applications of LMIIs 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 ($\mu$-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 $H^\infty$ control ([11], [14])
• Multi-objective $H^\infty$ 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 $A^T P + PA < 0$, $P > I$.

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:

$$A(t) \in \text{Co}\{A_1, \ldots, A_n\} = \left\{ \sum_{i=1}^{n} a_i A_i : a_i \geq 0, \sum_{i=1}^{N} a_i = 1 \right\}.$$

A sufficient condition for the asymptotic stability of this LDI is the feasibility of

Find $P = P^T$ such that $A_i^T P + PA_i < 0$, $P > I$. 
RMS Gain

The random-mean-squares (RMS) gain of a stable LTI system

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

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 \(\gamma\) over \(X = X^T\) and \(\gamma\) 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

\[
\|G\|_2^2 := \lim_{T \to \infty} \mathbb{E} \left\{ \frac{1}{T} \int_0^T y^T(t) y(t) dt \right\} = \frac{1}{2\pi} \int_{-\infty}^{\infty} G^H(j\omega)G(j\omega) d\omega.
\]
It can be shown that

$$\|G\|_2^2 = \inf \left\{ \text{Trace} \left( CPC^T \right) : AP + PA^T + BB^T < 0 \right\}.$$ 

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 $P,Q$).
Further Mathematical Background

Efficient interior-point algorithms are now available to solve the three generic LMI problems Equation 3-2–Equation 3-4 defined in “LMI Applications” on page 3-6. These algorithms have a polynomial-time complexity. That is, the number $N(\varepsilon)$ of flops needed to compute an $\varepsilon$-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 3-3 or the generalized eigenvalue minimization problem Equation 3-4.

Some LMI problems are formulated in terms of inequalities rather than strict inequalities. For instance, a variant of Equation 3-3 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

$$\begin{pmatrix} x & x \\ x & x \end{pmatrix} \geq 0.$$ 

Such problems cannot be handled directly by interior-point methods which require strict feasibility of the LMI constraints. A well-posed reformulation of Equation 3-5 would be

Minimize $c^T x$ subject to $x \geq 0$.

Keeping this subtlety in mind, we always use strict inequalities in this manual.
Bibliography


LMI Lab

- “Tools for Specifying and Solving LMIs” on page 4-2
- “Specifying a System of LMIs” on page 4-7
- “Specify LMI System at the Command Line” on page 4-8
- “Specify LMIs with the LMI Editor GUI” on page 4-15
- “How lmivar and lmiterm Manage LMI Representation” on page 4-18
- “Querying the LMI System Description” on page 4-19
- “LMI Solvers” on page 4-20
- “Minimize Linear Objectives under LMI Constraints” on page 4-22
- “Conversion Between Decision and Matrix Variables” on page 4-26
- “Validating Results” on page 4-27
- “Modify a System of LMIs” on page 4-28
- “Advanced LMI Techniques” on page 4-31
- “Bibliography” on page 4-42
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

\[ L(x) = L_0 + x_1L_1 + \ldots + x_NL_N < 0 \]

where

- \( L_0, L_1, \ldots, L_N \) are given symmetric matrices
- \( x = (x_1, \ldots, x_N)^T \in \mathbb{R}^N \) is the vector of scalar variables to be determined. We refer to \( x_1, \ldots, x_N \) 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

\[ A^T X + X A < 0 \]

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

\[
x_1 \begin{pmatrix} -2 & 2 \\ 2 & 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 & -3 \\ -3 & 4 \end{pmatrix} + x_3 \begin{pmatrix} 0 & 0 \\ 0 & -4 \end{pmatrix} < 0.
\]

Clearly this expression is less intuitive and transparent than Equation 4-1. Moreover, the number of matrices involved in Equation 4-2 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 \(A^T X + X A\) in the Lyapunov inequality Equation 4-1 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 \(H^\infty\) theory

\[
N^T \begin{pmatrix} A^T X + X A & X C^T & B \\ C X & -\gamma I & D \\ B^T & D^T & -\gamma I \end{pmatrix} N < 0
\]
where A, B, C, D, and N are given matrices and the problem variables are \( X = X^T \in \mathbb{R}^{n \times n} \) and \( \gamma \in \mathbb{R} \). We use the following terminology to describe such LMIs:

- \( N \) is called the outer factor, and the block matrix

\[
L(X, \gamma) = \begin{pmatrix}
A^T X + X A & X C^T & B \\
C X & -\gamma I & D \\
B^T & D^T & -\gamma I
\end{pmatrix}
\]

is called the inner factor. The outer factor needs not be square and is often absent.

- \( X \) and \( \gamma \) 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 \( \gamma \). This expression can be broken down into a sum of elementary terms. For instance, the block (1,1) contains two elementary terms: \( A^T X \) and \( X A \).

- 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 \( X A \), \( X C^T \), and \( -\gamma I \) above.

The LMI (Equation 4-3) 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 \( \gamma \), 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:
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 $\text{LMISYS}$ 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 $\text{LMISYS}$ 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 3-6. 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.
Specifying a System of LMIs

The LMI Lab can handle any system of LMIs of the form

\[ N^T L(X_1, \ldots, X_K) N < MT R(X_1, \ldots, X_K) M \]

where

- \( X_1, \ldots, X_K \) 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 \( X_1, \ldots, X_K \) 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 \( X_1, \ldots, X_K \).
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 \texttt{lmivar/lmiterm} commands that build it incrementally, or (2) via the LMI Editor \texttt{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 \texttt{lmivar} and \texttt{lmiterm}. Accordingly, beginners may elect to skip the subsections on \texttt{lmivar} and \texttt{lmiterm} and to concentrate on the GUI-based specification of LMIs with \texttt{lmiedit}.
Specify LMI System at the Command Line

The following tutorial example is used to illustrate the specification of LMI systems with the LMI Lab tools.

Specifying LMI System

Consider a stable transfer function

\[ G(s) = C(sI - A)^{-1} B \]

with four inputs, four outputs, and six states, and consider the set of input/output scaling matrices \( D \) with block-diagonal structure

\[
D = \begin{pmatrix}
d_1 & 0 & 0 & 0 \\
0 & d_1 & 0 & 0 \\
0 & 0 & d_2 & d_3 \\
0 & 0 & d_4 & d_5
\end{pmatrix}.
\]

The following problem arises in the robust stability analysis of systems with time-varying uncertainty [4]:

Find, if any, a scaling \( D \) of structure (Equation 4-5) such that the largest gain across frequency of \( D G(s) D^{-1} \) is less than one.

This problem has a simple LMI formulation: there exists an adequate scaling \( D \) if the following feasibility problem has solutions:

Find two symmetric matrices \( X \in R^{6x6} \) and \( S = D^T D \in R^{4x4} \) such that

\[
\begin{pmatrix}
A^T X + XA + C^T S C & XB \\
B^T X & -S
\end{pmatrix} < 0
\]

\[ X > 0 \]

\[ S > 1. \]
The LMI system (Equation 4-6, Equation 4-7, and Equation 4-8) can be described with the LMI Editor as outlined below. Alternatively, its internal description can be generated with `lmivar` and `lmiterm` commands as follows:

```matlab
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
```

Here the `lmivar` commands define the two matrix variables $X$ and $S$ while the `lmiterm` commands describe the various terms in each LMI. Upon completion, `getlmis` returns the internal representation `LMISYS` of this LMI system. The following subsections give more details on the syntax and usage of these various commands:

- “Initializing the LMI System” on page 4-9
- “Specifying the LMI Variables” on page 4-10
- “Specifying Individual LMIs” on page 4-12

More information on how the internal representation is updated by `lmivar/lmiterm` can also be found in “How lmivar and lmiterm Manage LMI Representation” on page 4-18.

**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

\texttt{setlmis(LMIS0)}

### Specifying the LMI Variables

The matrix variables are declared one at a time with \texttt{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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>Symmetric block diagonal structure. This corresponds to matrix variables of the form</td>
</tr>
</tbody>
</table>
|        | \[
|        | X = \begin{pmatrix}
|        | D_1 & 0 & \ldots & 0 \\
|        | 0 & D_2 & \ddots & \vdots \\
|        | \vdots & \ddots & \ddots & 0 \\
|        | 0 & \ldots & 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 |
|        | \( D_j = d \times I, \ d \in \mathbb{R} \) |
|        | 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, \( x_n \), or \(-x_n\) where \( x_n \) denotes the \( n \)-th decision variable in the problem. For details on how to use Type 3, see “Structured Matrix Variables” on page 4-31 as well as the \texttt{lmivar} entry in the reference pages. |

In “Specifying LMI System” on page 4-8, the matrix variables \( X \) and \( S \) are of Type 1. Indeed, both are symmetric and \( S \) inherits the block-diagonal structure from Equation 4-5 of \( D \). Specifically, \( S \) is of the form...
After initializing the description with the command `setlmis([]), these two matrix variables are declared by

\[
S = \begin{pmatrix}
  s_1 & 0 & 0 & 0 \\
  0 & s_1 & 0 & 0 \\
  0 & 0 & s_2 & s_3 \\
  0 & 0 & s_3 & s_4
\end{pmatrix}.
\]

In both 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

\[
\text{lmivar}(2, [3 \ 5])
\]

For convenience, `lmivar` also returns a “tag” that identifies the matrix variable for subsequent reference. For instance, \(X\) and \(S\) in “Specifying LMI System” on page 4-8 could be defined by

\[
X = \text{lmivar}(1, [6 \ 1]) \\
S = \text{lmivar}(1, [2 \ 0; 2 \ 1])
\]

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, \texttt{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 \texttt{lmiVar} entry in the reference pages for more details).

### Specifying Individual LMIs

After declaring the matrix variables with \texttt{lmiVar}, we are left with specifying the term content of each LMI. Recall that LMI terms fall into three categories:

- The \textit{constant terms}, i.e., fixed matrices like $I$ in the left side of the LMI $S > I$
- The \textit{variable terms}, i.e., terms involving a matrix variable. For instance, $A^TX$ and $C^TSC$ in Equation 4-6. Variable terms are of the form $PXQ$ where $X$ is a variable and $P$, $Q$ are given matrices called the left and right \textit{coefficients}, respectively.
- The \textit{outer factors}

The following rule should be kept in mind when describing the term content of an LMI:

\textbf{Note: } 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 \texttt{lmiterm}. For instance, the LMI

$$\begin{pmatrix}
A^TX +XA +C^TSC & XB \\
B^TX & -S
\end{pmatrix} < 0$$

is described by

\texttt{lmiterm([1 1 1 1],1,A,'s')}
\texttt{lmiterm([1 1 1 2],C','C')}
\texttt{lmiterm([1 1 2 1],1,B)}
\texttt{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 \texttt{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 \textit{real-valued}. See “Complex-Valued LMIs” on page 4-33 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 \(AXB + B^TX^T A^T\) with a single \texttt{lmiterm} command. For instance, the first command specifies \( A^T X + X A \) as the “symmetrization” of \( X A \). Finally, scalar values are allowed as shorthand for \textit{scalar matrices}, i.e., matrices of the form \( \alpha I \) with \( \alpha \) scalar. Thus, a constant term of the form \( \alpha I \) can be specified as the “scalar” \( \alpha \). 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 “Example: Specifying Matrix Variable Structures” on page 4-31 is described by

\[
\text{\texttt{lmiterm([[-3 \ 1 \ 1 \ 2],1,1]) \quad \% \ 1*S*1 = S}}
\]
\[
\text{\texttt{lmiterm([3 \ 1 \ 1 \ 0],1) \quad \% \ 1*I = I}}
\]

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 \texttt{lmivar} and the LMI identifiers are set by the function \texttt{newlmi}. These identifiers can be used in \texttt{lmiterm} commands to refer to a given LMI or matrix variable. For the LMI system of “Specifying LMI System” on page 4-8, this would look like:

\[
\text{\texttt{setlmis([])}}
\]
\[
\text{\texttt{X = lmivar(1,[6 1])}}
\]
\[
\text{\texttt{S = lmivar(1,[2 0;2 1])}}
\]
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, type

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.
Specify LMIs with the LMI Editor GUI

The LMI Editor \texttt{lmiedit} is a graphical user interface (GUI) to specify LMI systems in a straightforward symbolic manner. Typing \texttt{lmiedit}

calls up a window with several editable text areas and various buttons.

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 \texttt{lmivar} (see “Specifying the LMI Variables” on page 4-10 for details).

Please use \textit{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

\[
\begin{pmatrix}
A^T X + X A & X B \\
B^T X & -I
\end{pmatrix} < 0
\]

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 \textit{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 \texttt{lmivar}/\texttt{lmiterm} commands needed to describe this LMI system (\texttt{view commands} button). Conversely, the LMI system defined by a particular sequence of \texttt{lmivar}/\texttt{lmiterm} commands can be displayed as a MATLAB expression by clicking on the \texttt{describe...} buttons.

Beginners can use this facility as a tutorial introduction to the \texttt{lmivar} and \texttt{lmiterm} commands.

- Save the symbolic description of the LMI system as a MATLAB string (\texttt{save} button). This description can be reloaded later on by clicking the \texttt{load} button.

- Read a sequence of \texttt{lmivar}/\texttt{lmiterm} commands from a file (\texttt{read} button). You can then click on \texttt{describe the matrix variables} or \texttt{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 \texttt{lmivar}/\texttt{lmiterm} commands needed to describe a particular LMI system (\texttt{write} button).
Specify LMIswith the LMI Editor GUI

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, lmi edit 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
  
  \[(a*x+b)\cdot c + c\cdot(a*x+b)\]

  is invalid when x is a variable name. By contrast,

  \[(a+b)\cdot x + x\cdot(a+b)\]

  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.
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`
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)
```
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 $x \in \mathbb{R}^N$ (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 $x \in \mathbb{R}^N$ subject to $A(x) < B(x)$

  The corresponding solver is called `mincx`.

- **Generalized eigenvalue minimization problem**

  Minimize $\lambda$ over $x \in \mathbb{R}^N$ subject to

  $$C(x) < D(x)$$

  $$0 < B(x)$$

  $$A(x) < \lambda B(x).$$

  The corresponding solver is called `gevp`.

Note that $A(x) < B(x)$ 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 of Nesterov and Nemirovski [3], [2].
For generalized eigenvalue minimization problems, it is necessary to distinguish between the standard LMI constraints $C(x) < D(x)$ and the linear-fractional LMIs $A(x) < \lambda B(x)$ attached to the minimization of the generalized eigenvalue $\lambda$. When using \texttt{gevp}, you should follow these three rules to ensure proper specification of the problem:

- Specify the LMIs involving $\lambda$ as $A(x) < B(x)$ \textit{(without the $\lambda$)}
- Specify them \textit{last} in the LMI system. \texttt{gevp} systematically assumes that the last $L$ LMIs are linear-fractional if $L$ is the number of LMIs involving $\lambda$
- 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 \texttt{gevp}.

An initial guess $x_{init}$ for $x$ can be supplied to \texttt{mincx} or \texttt{gevp}. Use \texttt{mat2dec} to derive $x_{init}$ from given values of the matrix variables $X_1, \ldots, X_K$.

The following example illustrates the use of the \texttt{mincx} solver.
Minimize Linear Objectives under LMI Constraints

Consider the optimization problem:

Minimize Trace($X$) subject to

$$A^TX +XA + XBB^TX + Q < 0$$

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 4-9 is equivalent to the following linear objective minimization problem:

Minimize Trace($X$) subject to

$$\begin{pmatrix} A^TX +XA + Q & XB \\ B^TX & -I \end{pmatrix} < 0.$$ 

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 4-9 by the sequence of commands

```matlab
setlmis([])
X = lmivar(1,[3 1]) % variable X, full symmetric
lmiterm([1 1 1 X],1,a,'s')
lmiterm([1 1 0],q)
```
lmiterm([1 2 2 0],-1)
lmiterm([1 2 1 X],b',1)

LMIs = getlmis

2 Write the objective $\text{Trace}(X)$ as $c^T x$ 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,

$$c = \text{mat2dec}(\text{LMIs}, \text{eye}(3))$$

Note that the function \text{defcx} provides a more systematic way of specifying such objectives (see “Specifying $c^T x$ Objectives for \text{mincx}” on page 4-36 for details).

3 Call \text{mincx} to compute the minimizer $x_{\text{opt}}$ and the global minimum $c_{\text{opt}} = c' x_{\text{opt}}$ of the objective:

$$\text{options} = [1e-5,0,0,0,0]$$
$$[c_{\text{opt}}, x_{\text{opt}}] = \text{mincx}(\text{LMIs}, c, \text{options})$$

Here $1e-5$ specifies the desired relative accuracy on $c_{\text{opt}}$.

The following trace of the iterative optimization performed by \text{mincx} appears on the screen:

Solver for linear objective minimization under LMI constraints
Iterations : Best objective value so far

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-8.511476</td>
</tr>
<tr>
<td>3</td>
<td>-13.063640</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound: -34.023978</td>
</tr>
<tr>
<td>4</td>
<td>-15.768450</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound: -25.005604</td>
</tr>
<tr>
<td>5</td>
<td>-17.123012</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound: -21.306781</td>
</tr>
<tr>
<td>6</td>
<td>-17.882558</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound: -19.819471</td>
</tr>
<tr>
<td>7</td>
<td>-18.339853</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------</td>
</tr>
<tr>
<td>8</td>
<td>-18.552558</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>9</td>
<td>-18.646811</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>10</td>
<td>-18.687324</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>11</td>
<td>-18.705715</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>12</td>
<td>-18.712175</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>13</td>
<td>-18.714880</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>14</td>
<td>-18.716094</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>15</td>
<td>-18.716509</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
<tr>
<td>16</td>
<td>-18.716695</td>
</tr>
<tr>
<td>***</td>
<td>new lower bound:</td>
</tr>
</tbody>
</table>

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 4-10) 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, \texttt{mincx} reports that the global minimum for the objective Trace($X$) is $-18.716695$ with relative accuracy of at least $9.5 \times 10^{-6}$. This is the value $c_{opt}$ returned by \texttt{mincx}.

\texttt{mincx} also returns the optimizing vector of decision variables $x_{opt}$. The corresponding optimal value of the matrix variable $X$ is given by

$$X_{opt} = \text{dec2mat}(\text{LMIs}, x_{opt}, X)$$

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 \texttt{care}:

$$X_{st} = \text{care}(a,b,q,-1)$$
$$\text{norm}(X_{opt} - X_{st})$$

$$\text{ans} = 6.5390e-05$$
Conversion Between Decision and Matrix Variables

While LMIs are specified in terms of their matrix variables $X_1, \ldots, X_K$, 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 $X_1, X_2, X_3$ of these variables, the corresponding value $x_{\text{dec}}$ of the vector of decision variables is returned by `mat2dec`:

$$x_{\text{dec}} = \text{mat2dec}(\text{LMISYS}, X_1, X_2, X_3)$$

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 $x_{\text{dec}}$ of the vector of decision variables, the corresponding value of the $k$-th matrix is given by `dec2mat`. For instance, the value $X_2$ of the second matrix variable is extracted from $x_{\text{dec}}$ by

$$X_2 = \text{dec2mat}(\text{LMISYS}, x_{\text{dec}}, 2)$$

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.
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 4-22, you can verify that the minimizer `xopt` returned by `mincx` satisfies the LMI constraint (Equation 4-10) as follows:

```matlab
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

```matlab
eig(lhs-rhs)
```

```matlab
an =
-2.0387e–04
-3.9333e–05
-1.8917e–07
-4.6680e+01
```
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 “Specifying LMI System” on page 4-8 is described in LMISYS and that we want to remove the positivity constraint on $X$. This is done by

$$\text{NEWSYS} = \text{dellmi}(\text{LMISYS},2)$$

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, Equation 4-6–Equation 4-8, Slmi keeps pointing to $S > I$ even after deleting the second LMI by

$$\text{NEWSYS} = \text{dellmi}(\text{LMISYS}, \text{Xpos})$$

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

$$\text{setlmis}([])$$
\( X = \text{lmivar}(1,[4 1]) \) % X
\( W = \text{lmivar}(2,[2 4]) \) % W

\( \text{lmiterm}([1 1 1 X],1,A,'s') \)
\( \text{lmiterm}([1 1 1 W],B,1,'s') \)
\( \text{lmiterm}([1 1 1 0],1) \)

LMISYS = getlmis

To delete the variable \( W \), type the command

NEWSYS = \text{delmvar}(\text{LMISYS},W)

The resulting \( \text{NEWSYS} \) now describes the Lyapunov inequality

\( A^T X + XA + I < 0 \)

Note that \text{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 \text{setmvar}.

**Instantiating a Matrix Variable**

The function \text{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 \text{setmvar} some variables and optimize with respect to the remaining ones.

Consider again “Specifying LMI System” on page 4-8 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 \( S = 2-\beta \psi I \). 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

$$
\begin{pmatrix}
A^T X + XA + 2C^{TC} & XB \\
B^T X & -2I
\end{pmatrix} < 0
$$

with $X > 0$ and $2I > I$.

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.
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 4-2.

Structured Matrix Variables

Fairly complex matrix variable structures and interdependencies can be specified with \texttt{lmivar}. Recall that the symmetric block-diagonal or rectangular structures are covered by Types 1 and 2 of \texttt{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 \texttt{lmivar}. First, consider the case of uncorrelated matrix variables.

**Example: Specifying Matrix Variable Structures**

Suppose that the problem variables include a 3-by-3 symmetric matrix $X$ and a 3-by-3 symmetric Toeplitz matrix:

$$
Y = \begin{pmatrix}
  y_1 & y_2 & y_3 \\
  y_2 & y_1 & y_2 \\
  y_3 & y_2 & y_1 
\end{pmatrix}.
$$

The variable $Y$ has three independent entries, hence involves three decision variables. Since $Y$ is independent of $X$, these decision variables should be labeled $n + 1$, $n + 2$, $n + 3$ where $n$ is the number of decision variables involved in $X$. To retrieve this number, define the variable $X$ (Type 1) by

\begin{verbatim}
setlmis([])
[X,n] = lmivar(1,[3 1])
\end{verbatim}

The second output argument $n$ gives the total number of decision variables used so far (here $n = 6$). Given this number, $Y$ can be defined by

\begin{verbatim}
Y = lmivar(3,n+[1 2 3;2 1 2;3 2 1])
\end{verbatim}
or equivalently by

\[ Y = \text{lmivar}(3, \text{toeplitz}(n+\[1 \ 2 \ 3\])) \]

where \text{toeplitz} is a standard MATLAB function. For verification purposes, we can visualize the decision variable distributions in \(X\) and \(Y\) with \text{decinfo}:

\[
\text{lmis} = \text{getlmis} \\
\text{decinfo(lmis,}X) \\
\text{ans} = \\
1 \ 2 \ 4 \ \\
2 \ 3 \ 5 \ \\
4 \ 5 \ 6 \\

decinfo(lmis,}Y) \\
\text{ans} = \\
7 \ 8 \ 9 \ \\
8 \ 7 \ 8 \ \\
9 \ 8 \ 7 \\
\]

The next example is a problem with interdependent matrix variables.

**Example: Specifying Interdependent Matrix Variables**

Consider three matrix variables \(X\), \(Y\), \(Z\) with 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 \(x, y, z, t\) are independent scalar variables. To specify such a triple, first define the two independent variables \(X\) and \(Y\) (both of Type 1) as follows:

\[
\text{setlmi}([]) \\
[X,n,\text{sX}] = \text{lmivar}(1,[1 \ 0;1 \ 0]) \\
[Y,n,\text{sY}] = \text{lmivar}(1,[1 \ 0;1 \ 0])
\]

The third output of \text{lmivar} gives the entry-wise dependence of \(X\) and \(Y\) on the decision variables \((x_1, x_2, x_3, x_4) := (x, y, z, t)\):

\[
\text{sX} =
\]
Using Type 3 of `lmivar`, you can now specify the structure of $Z$ in terms of the decision variables $x_1 = x$ and $x_4 = t$:

$$[Z, n, sZ] = \text{lmivar}(3, [0 \ -sX(1,1); -sY(2,2) \ 0])$$

Since $sX(1,1)$ refers to $x_1$ while $sY(2,2)$ refers to $x_4$, this defines the variable $Z$ as:

$$Z = \begin{pmatrix} 0 & -x_1 \\ -x_4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -x \\ -t & 0 \end{pmatrix}$$

as confirmed by checking its entry-wise dependence on the decision variables:

$$sZ = \begin{pmatrix} 0 & -1 \\ -4 & 0 \end{pmatrix}$$

### 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

$$\begin{pmatrix} \text{Re}(L(x)) & \text{Im}(L(x)) \\ -\text{Im}(L(x)) & \text{Re}(L(x)) \end{pmatrix} < 0.$$ 

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

$$M^HXM < X, \quad X = X^H > I$$

reads (given the decompositions $M = M_1 + jM_2$ and $X = X_1 + jX_2$ with $M_j$, $X_j$ 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 \( M \in \mathbb{C}^{5 \times 5} \), the LMI system (Equation 4-11) would be specified as follows:

\[
M_1 = \text{real}(M), \quad M_2 = \text{imag}(M) \\
bigM = [M_1 \ M_2; -M_2 \ M_1] \\
\text{setlmis}([])
\]

% declare \( \text{bigX} = [X_1 \ X_2; -X_2 \ X_1] \) with \( X_1 = X_1' \) and \( X_2 + X_2' = 0 \):

\[
[X_1,n_1,sX_1] = \text{lmivar}(1, [5 \ 1]) \\
[X_2,n_2,sX_2] = \text{lmivar}(3, \text{skewdec}(5,n_1)) \\
bigX = \text{lmivar}(3, [sX_1 \ sX_2; -sX_2 \ sX_1])
\]

% describe the real counterpart of (1.12):

\[
\text{lmiterm}([1 \ 1 \ 1 \ 0],1) \\
\text{lmiterm}([-1 \ 1 \ 1 \ \text{bigX}],1,1) \\
\text{lmiterm}([2 \ 1 \ 1 \ \text{bigX}],\text{bigM}',\text{bigM}) \\
\text{lmiterm}([-2 \ 1 \ 1 \ \text{bigX}],1,1)
\]

\[
\text{lmis} = \text{getlmis}
\]

Note the three-step declaration of the structured matrix variable \( \text{bigX} \),

\[
\text{bigX} = \begin{pmatrix} X_1 & X_2 \\ -X_2 & X_1 \end{pmatrix}.
\]

1. Specify \( X_1 \) as a (real) symmetric matrix variable and save its structure description \( sX_1 \) as well as the number \( n_1 \) of decision variables used in \( X_1 \).

2. Specify \( X_2 \) as a skew-symmetric matrix variable using Type 3 of \text{lmivar} and the utility \text{skewdec}. The command \text{skewdec}(5,n_1) creates a 5-by-5 skew-symmetric structure depending on the decision variables \( n_1 + 1, n_1 + 2, \ldots \)

3. Define the structure of \( \text{bigX} \) in terms of the structures \( sX_1 \) and \( sX_2 \) of \( X_1 \) and \( X_2 \).

See “Structured Matrix Variables” on page 4-31 for more details on such structure manipulations.
Specifying $c^T x$ Objectives for mincx

The LMI solver mincx minimizes linear objectives of the form $c^T x$ 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 $\text{Trace}(X)$ where $X$ is a symmetric matrix variable, or $u^T X u$ 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

$$\text{Trace}(X) + x_0^T Px_0$$

where $X$ and $P$ are two symmetric variables and $x_0$ is a given vector. If lmisys is the internal representation of the LMI system and if $x_0$, $X$, $P$ have been declared by

```matlab
x0 = [1;1]
selmis([])
X = lmivar(1,[3 0])
P = lmivar(1,[2 1])
:
:
lmisys = getlmis
```

the $c$ vector such that $c^T x = \text{Trace}(X) + x_0^T Px_0$ can be computed as follows:

```matlab
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 $x_j = 1$. 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 = 10^9`. 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

\[ L(x) \leq 0 \]

has solutions while

\[ L(x) < 0 \]

has no solution.

For feasibility problems, this difficulty is automatically circumvented by \texttt{feasp}, which reformulates the problem:

Find \( x \) such that

\[ L(x) \leq 0 \]

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 4-12 is feasible if and only if the global minimum \( t_{\text{min}} \) of Equation 4-12 satisfies

\[ t_{\text{min}} \leq 0 \]

For feasible but not strictly feasible problems, however, the computational effort is typically higher as \texttt{feasp} strives to approach the global optimum \( t_{\text{min}} = 0 \) to a high accuracy.

For the LMI problems addressed by \texttt{mincx} and \texttt{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

\[ A^T P + PA < 0, \ P > 0 \]

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 \( \lambda \) subject to

\[ A(x) < \lambda B(x), \ B(x) > 0, \ C(x) < 0. \]

Technically, the positivity of \( B(x) \) for some \( x \in \mathbb{R}^n \) 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

\[ A(x) < \begin{pmatrix} Y & 0 \\ 0 & 0 \end{pmatrix}, \ Y < \lambda B_1(x), \ B_1(x) > 0 \]

where \( Y \) is an additional symmetric variable of proper dimensions. The resulting problem
is equivalent to Equation 4-14 and can be solved directly with gevp.
Efficiency and Complexity Issues

As explained in “Tools for Specifying and Solving LMIs” on page 4-2, the term-oriented description of LMIs used in the LMI Lab typically leads to higher efficiency than the canonical representation

\[ A_0 + x_1A_1 + ... + x_NA_N < 0. \]

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 4-15 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. Cholesly 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 \( M + P^TXQ + Q^TX^TP < 0 \)

In many output-feedback synthesis problems, the design can be performed in two steps:

2. Given this Lyapunov function, derive the controller state-space matrices by solving an LMI of the form

\[ M + P^TXQ + Q^TX^TP < 0 \]

where \( M, P, Q \) are given matrices and \( X \) is an unstructured \( m \)-by-\( n \) matrix variable.
It turns out that a particular solution $X_c$ of Equation 4-16 can be computed via simple linear algebra manipulations [1]. Typically, $X_c$ corresponds to the center of the ellipsoid of matrices defined by Equation 4-16.

The function `basiclmi` returns the “explicit” solution $X_c$:

$$X_c = \text{basiclmi}(M,P,Q)$$

Since this central solution sometimes has large norm, `basiclmi` also offers the option of computing an approximate least-norm solution of Equation 4-16. This is done by

$$X = \text{basiclmi}(M,P,Q,'Xmin')$$

and involves LMI optimization to minimize $\|X\|$.
Bibliography


Analyzing Uncertainty Effects in Simulink

- “Analyzing Uncertainty in Simulink” on page 5-2
- “Simulink Blocks for Analyzing Uncertainty” on page 5-3
- “Specify Uncertainty Using Uncertain State Space Blocks” on page 5-4
- “Simulate Uncertainty Effects” on page 5-7
- “Vary Uncertainty Values Using Individual Uncertain State Space Blocks” on page 5-8
- “Vary Uncertainty Values Across Multiple Uncertain State Space Blocks” on page 5-14
- “Computing Uncertain State-Space Models from Simulink Models” on page 5-19
- “Obtain Uncertain State-Space Model from Simulink Model” on page 5-20
- “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 5-21
- “Linearize Block to Uncertain Model” on page 5-24
- “Using Uncertain Linearization for Analysis or Control Design” on page 5-26
- “Analyzing Stability Margin of Simulink Models” on page 5-27
- “Stability Margin of a Simulink Model” on page 5-29
Analyzing Uncertainty in Simulink

Robust Control Toolbox software provides tools to model uncertainty in Simulink®. Using these tools, you can analyze how the uncertainty impacts the time- and frequency-domain behavior of the 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 5-4. 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 5-7.
- 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 5-7 and “Obtain Uncertain State-Space Model from Simulink Model” on page 5-20.
- 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 5-20.

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 5-21. 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, on the Home tab, click Simulink Library to launch to Library Browser. In the Library Browser, select Robust Control Toolbox.
Specify Uncertainty Using Uncertain State Space Blocks

- “How to Specify Uncertainty in Uncertain State Space Blocks” on page 5-4
- “Next Steps” on page 5-6

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 5-3.
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.
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` and `ultidyn`. The model depends on a set of uncertain variables (`ureal` 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.

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 5-7.

(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 5-7.
- Perform an uncertain linearization, as described in “Obtain Uncertain State-Space Model from Simulink Model” on page 5-20.
Simulate Uncertainty Effects

- “How to Simulate Effects of Uncertainty” on page 5-7
- “How to Vary Uncertainty Values” on page 5-7

How to Simulate Effects of Uncertainty

As described in “Specify Uncertainty Using Uncertain State Space Blocks” on page 5-4, the uncertain state-space model you specify in the Uncertain State Space block depends on a set of uncertain variables (ureal 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 5-7. 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 “Vary Uncertainty Values Using Individual Uncertain State Space Blocks” on page 5-8.
- 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 Uncertainty Values Across Multiple Uncertain State Space Blocks” on page 5-14.
Vary Uncertainty Values Using Individual Uncertain State Space Blocks

This example shows the workflow for varying uncertainty values using individual Uncertain State Space blocks in a Simulink model. Use this approach for simple models with few uncertain variables or one Uncertain State Space block.

This section uses a simple Simulink model to provide step-by-step instructions for:

• Toggling between nominal, manually defined and randomly-generated uncertainty values associated with the Uncertain State Space block.
• Simulating the model's responses for these uncertainty values.

1. Open the Simulink model rct_sim_ex1.

   rct_sim_ex1

   The model contains an Uncertain State Space block called Uncertain System, as shown in the following figure.

   ![Simulink model diagram](image)

2. Double-click the Uncertain System block to open the Function Block Parameters dialog box.
The Uncertain System block uses the uncertain system variable `unc_sys`. This variable is defined in the Model workspace as `unc_sys=`
ss(ureal('a',-1,'Range',[-2 -.5]),1,5,0)*(1+0.1*input_unc). The uncertain model depends on a single uncertain variable named a. The **Uncertainty value** field specifies to use nominal value of the uncertain variable a.

Click **OK** to close the dialog box.

3 Click to simulate the model.

The software uses the nominal value of a during simulation. After the simulation completes, the **MultiPlot Graph** shows the following plot.

4 To simulate the model using a manually defined value of a:

   a Double-click the **Uncertain State Space** block, and enter `struct('a',-0.3)` in the **Uncertainty value** field.
Click to simulate the model.

The MultiPlot Graph shows the following responses, corresponding to the nominal and manually-defined values of $a$. 
Pick a random value of \( a \) in its uncertainty range. To do so, double-click the Uncertain State Space block, and type `usample(ufind(unc_sys))` in the Uncertainty value field.

Analyze the uncertainty effects by simulating the model ten times using the following commands:
for i=1:10;
    sim('rct_sim_ex1',10);
end

During simulation, the software samples the uncertain variable $a$ in its uncertainty range $[-2, -0.5]$ and shows the simulated response for each sample value. The plots cycle through seven different colors, and the last response appears in red.

**Tip:** You can clear the plots in the MultiPlot Graph block before you run the simulation.
Vary Uncertainty Values Across Multiple Uncertain State Space Blocks

This example shows the workflow for varying uncertainty values across multiple Uncertain State Space blocks in a Simulink model. Use this approach for complex models with a large number of uncertain variables or Uncertain State Space blocks.

This section uses a Simulink model to provide step-by-step instructions for toggling between nominal and user-defined uncertainty values at the MATLAB prompt.

1. Open the Simulink model `rct_sim_ex2`.

   `rct_sim_ex2`

   The model contains two Uncertain State Space blocks, as shown in the following figure.

   ![Simulink model diagram](image)

   The Unmodeled dynamics and First order with uncertain pole blocks depend on the uncertain variables `input_unc` and `a`.

2. Double-click the Unmodeled dynamics block to open the block parameters dialog box. The Uncertainty value field contains the variable `val_all`. Similarly, the Uncertainty value field in the First order with uncertain pole block parameters dialog contains the variable `val_all`. You use this variable to vary the uncertain variable values across both the Uncertain State Space blocks.
Vary Uncertainty Values Across Multiple Uncertain State Space Blocks

**Note** When defining `val_all`, you can enter only a subset of uncertain variables referenced by the model in the structure. When you do not specify some uncertain variables, the software uses their nominal value during simulation.

3 At the MATLAB prompt, specify `val_all = []`; and click ![](play_button.png) to simulate the model.

The software uses the nominal values of the uncertain variables `a` and `input_unc` during simulation. After the simulation completes, the MultiPlot Graph block shows the following figure.
4 Generate random samples of uncertainty values:

- Find all Uncertain State Space blocks and associated uncertain variables in the model.

\[
\text{uvars=ufind('rct_sim_ex2')}\]

MATLAB returns the following result:

\[
\text{uvars} = \\
\quad a: [1x1 \text{ ureal}] \\
\quad \text{input_unc: [1x1 ultidyn]}
\]
b Randomly sample the uncertain variables.

\[
\text{val\_all} = \text{usample(uvars)}
\]

MATLAB returns the following result:

\[
\text{val\_all} = \\
\begin{align*}
\text{a:} & \quad -1.1167 \\
\text{input\_unc:} & \quad [1\times1 \text{ ss}]
\end{align*}
\]

The structure `val_all` contains sample values of the uncertain variables `a` and `input_unc`. The software samples the values within the specified uncertainty ranges for `a` and `input_unc`.

5 Simulate the model for the uncertainty values `val_all`. By repeating the process inside a for-loop, you can assess how uncertainty affects the model responses. For example, perform 10 simulations using random uncertainty values:

```
for i=1:10;
    \text{val\_all} = \text{usample(uvars)}
    \text{sim('rct_sim_ex2',10)};
end
```

During each simulation, the software samples values of the uncertain variables `input_unc` and `a` and plots the response for the sampled values. The MultiPlot Graph block shows the following responses obtained using random sample values of uncertain variables.
Analyzing Uncertainty Effects in Simulink
Computing 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 5-20.
- Using the Simulink Control Design linearize command, as described in “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 5-21.
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 5-21.

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 “Specifying Portion of Model to Linearize” in the Simulink Control Design documentation.
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 5-4.

**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” in the Simulink Control Design documentation.

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 “Controlling Block Linearization” in the Simulink Control Design documentation.

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 5-20.

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.
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 “Controlling Block Linearization” in the Simulink Control Design documentation.

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** or **uss**.

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 Block to Uncertain Model” on page 5-24.

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” in the Simulink Control Design documentation.

For an example of how to use the **linearize** command to compute an uncertain linearization, see Linearization of Simulink Models with Uncertainty.
Linearize Block to Uncertain Model

This example demonstrates how to specify a core Simulink block to linearize to an uncertain variable and compute an uncertain linearization at the command line. To learn how to perform this task using the user interface, see “Specify Uncertain Linearization for Core or Custom Simulink Blocks” on page 5-21.

1. Open the Simulink model.
   ```matlab
   slexAircraftExample
   ```

2. Double-click the Aircraft Dynamics Model block to view the subsystem.

3. Specify uncertain variables for the gain blocks $Z_d$ and $M_w$ in the Aircraft Dynamics Model subsystem.
   ```matlab
   spec.Type = 'Expression';
   spec.Specification = ureal('Zd',-63.9979,'Percentage',20);
   BlockSubs(1) = struct('Name','slexAircraftExample/Aircraft Dynamics Model/Gain5','Value',spec);
   ```
spec.Specification = ureal('Mw',-63.9979,'Percentage',20);
BlockSubs(2) = struct('Name','slexAircraftExample/Aircraft Dynamics Model/Gain4','Value',spec);

4 Compute the uncertain linearization.

mdl = 'slexAircraftExample';
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 = -64, variability = [-20,20]%, 1 occurrences
- Zd: Uncertain real, nominal = -64, variability = [-20,20]%, 1 occurrences

Type "sys.NominalValue" to see the nominal value, "get(sys)" to see all properties, and "sys.Uncertainty" to interact with the uncertain elements.

sys is an uncertain state-space (uss) model.
Using Uncertain Linearization for Analysis or Control Design

After computing an uncertain linearization, you can perform one of the following:

- Perform robustness analysis.
- Perform robust control design.
Analyzing Stability Margin of Simulink Models

Robust Control Toolbox provides the `loopmargin` command to analyze the stability margins of LTI models created in MATLAB and Simulink models. To use `loopmargin` with Simulink models, you must have the Simulink Control Design software. This section describes the difference between the MATLAB and Simulink approaches of using `loopmargin` and the workflow for computing the stability margin of Simulink models. For more information on how to analyze the stability margins of LTI models, see “MIMO Robustness Analysis”.

How Stability Margin Analysis Using Loopmargin Differs Between Simulink and LTI Models

When analyzing stability margins of LTI models using the syntax `[cm, dm, mm] = loopmargin(P, C)`, the software assumes the input and output of the linear plant `P` as the margin analysis points, as shown in the following figure.

Analyzing stability margin of Simulink models differs from analyzing stability margin of LTI models because you can enter specific margin analysis points in the Simulink model. For more information on how to assign margin analysis points in Simulink models, see the “Usage with Simulink” section of the `loopmargin` reference page.

Stability Margin of Simulink Model

The `loopmargin` command computes the following types of stability margins:
• Loop-at-a-time classical gain and phase margins
• Loop-at-a-time disk margins
• Multi-loop disk margin

To learn more about these stability margins, see the “Algorithms” section of the loopmargin reference page.

The loopmargin command computes the stability margin based on linearization of Simulink models. To compute stability margins of a Simulink model:

1 Specify the block where you want to define a margin analysis point.
2 Specify the output port of the block where you want the margin analysis point.

   The software performs the analysis by opening the loop at all specified margin analysis point.
3 Use the loopmargin command to compute the stability margins at the margin analysis point.

Optionally, you can compare the classical gain and phase margins obtained using loopmargin with the stability margins computed for the linearized model. The results using the two approaches should match for simple SISO models. For MIMO models, the loopmargin command provides richer robustness information. For an example, see “Stability Margin of a Simulink Model” on page 5-29.

Additionally, you can compute stability margins by specifying multiple margin analysis points and multiple operating points. For an example, see Loop Margins for an Airframe Autopilot.
Stability Margin of a Simulink Model

This example illustrates how to compute the stability margins of the `airframemargin` model and compare the results with stability margins computed using the linearized model.

1. Open the Simulink model:
   ```matlab
   airframemargin
   ```

2. Define a margin analysis point at the output of the Airframe Model block.
   ```matlab
   block1 = 'airframemargin/Airframe Model';
   ```

3. Specify the output `az` of the Airframe Model block as a margin analysis point.
   ```matlab
   port1 = 1;
   ```

   ```matlab
   [cm, dm, mm] = loopmargin('airframemargin', block1, port1);
   ```

5. View the classical gain and phase margins.
   ```matlab
   cm
   ```
6 Compare the classical gain and phase margins cm with stability margins of the linearized model computed using allmargin:

```matlab
% Define linearization I/O points.
io = linio('airframemargin/Airframe Model',1,'looptransfer');
% Linearize the model.
lin_sys = linearize('airframemargin',io);
% Compute gain and phase margins.
cm_lin = allmargin(-lin_sys)
```

```matlab
cm_lin =

GainMargin: [4.5652 2.5055e+003]
GMFrequency: [7.1979 314.1593]
PhaseMargin: 65.1907
PMFrequency: 2.1463
DelayMargin: 53.0113
DMFrequency: 2.1463
Stable: 1
```

The gain and phase margins, cm and cm_lin, computed using the two approaches match.