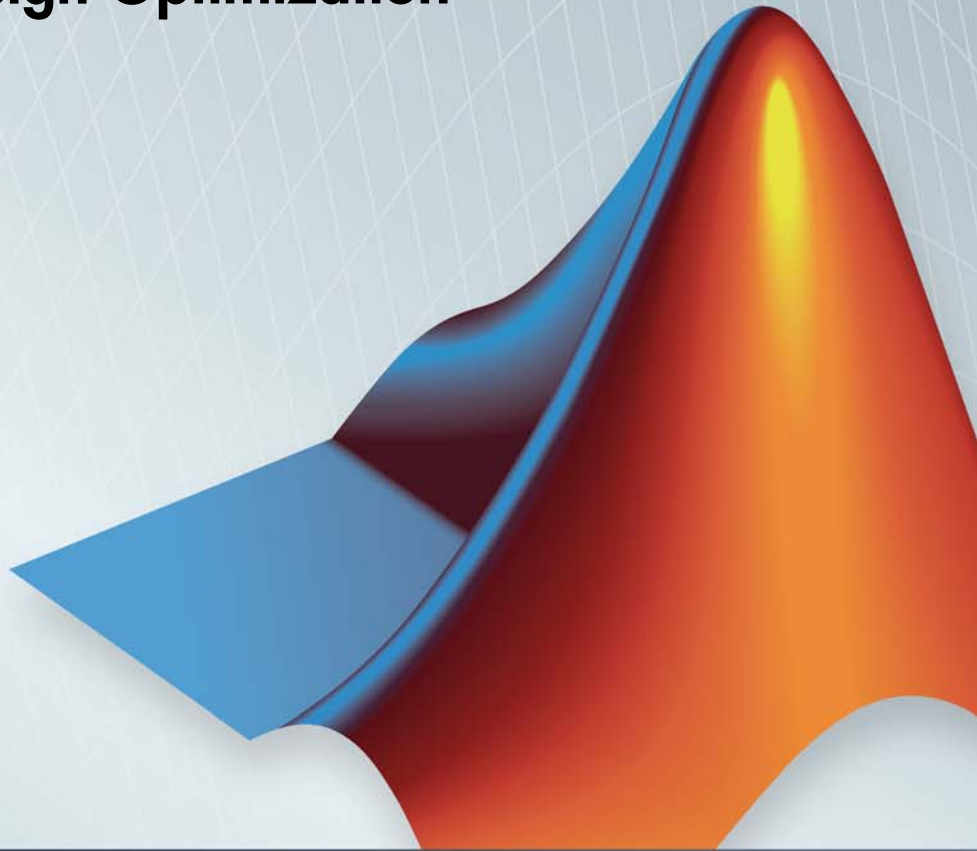


Simulink® Design Optimization™ Reference

R2014a



MATLAB® & SIMULINK®



How to Contact MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Simulink® Design Optimization™ Reference

© COPYRIGHT 1998–2014 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

September 2011 Online only
March 2012 Online only
September 2012 Online only
March 2013 Online only
September 2013 Online only
March 2014 Online only

New for Version 2.0 (Release R2011b)
Revised for Version 2.1 (Release R2012a)
Revised for Version 2.2 (Release R2012b)
Revised for Version 2.3 (Release R2013a)
Revised for Version 2.4 (Release R2013b)
Revised for Version 2.5 (Release 2014a)

Blocks — Alphabetical List

1

Class Reference

2

Alphabetical List

3

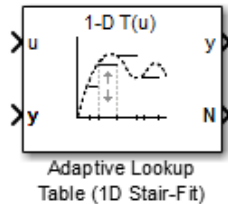
Blocks — Alphabetical List

Adaptive Lookup Table (1D Stair-Fit)

Purpose One-dimensional adaptive table lookup

Library Simulink Design Optimization

Description



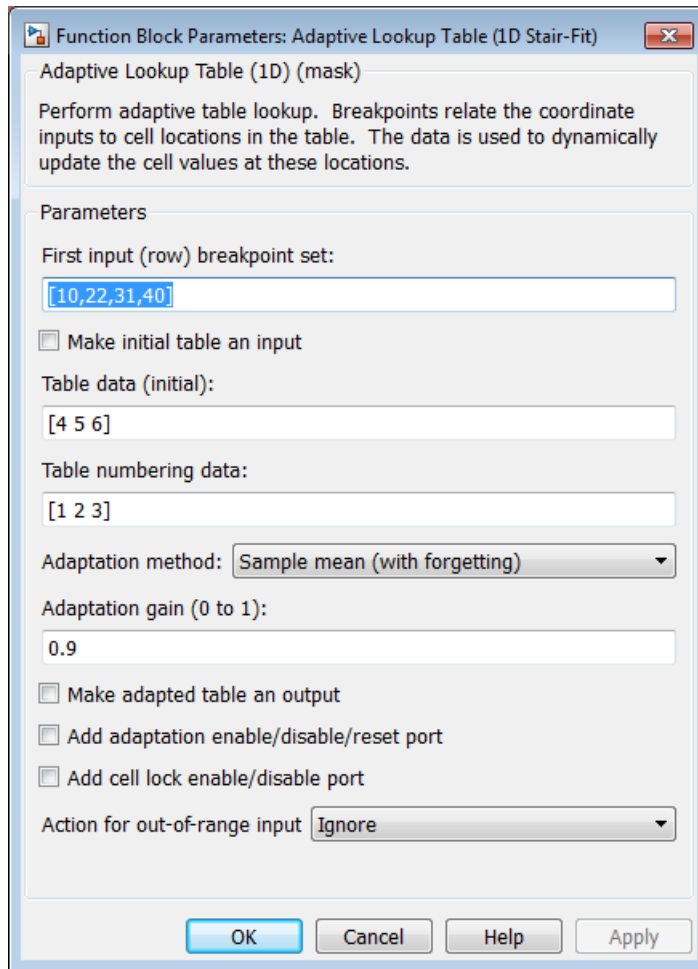
The Adaptive Lookup Table (1D Stair-Fit) block creates a one-dimensional adaptive lookup table by dynamically updating the underlying lookup table. The block uses the outputs, y , of your system to do the adaptations.

Each indexing parameter u may take a value within a set of adapting data points, which are called *breakpoints*. Two breakpoints in each dimension define a *cell*. The set of all breakpoints in one of the dimensions defines a *grid*. In the one-dimensional case, each cell has two breakpoints, and the cell is a line segment.

You can use the Adaptive Lookup Table (1D Stair Fit) block to model time-varying systems with one input.

Data Type Support Doubles only

Adaptive Lookup Table (1D Stair-Fit)



Dialog Box

First input (row) breakpoint set

The vector of values containing possible block input values. The input vector must be monotonically increasing.

Adaptive Lookup Table (1D Stair-Fit)

Make initial table an input

Selecting this check box forces the Adaptive Lookup Table (1D Stair-Fit) block to ignore the **Table data (initial)** parameter, and creates a new input port T_{in} . Use this port to input the table data.

Table data (initial)

The initial table output values. This vector must be of size $N-1$, where N is the number of breakpoints.

Table numbering data

Number values assigned to cells. This vector must be the same size as the table data vector, and each value must be unique.

Adaptation method

Choose `Sample mean` or `Sample mean (with forgetting)`. `Sample mean` averages all the values received within a cell. `Sample mean with forgetting` gives more weight to the new data. How much weight is determined by the **Adaptation gain** parameter. For more information, see “Selecting an Adaptation Method”.

Adaptation gain (0 to 1)

A number between 0 and 1 that regulates the weight given to new data during the adaptation. A 0 means short memory (last data becomes the table value), and 1 means long memory (average all data received in a cell).

Make adapted table an output

Selecting this check box creates an additional output port T_{out} for the adapted table.

Add adaptation enable/disable/reset port

Selecting this check box creates an additional input port `Enable` that enables, disables, or resets the adaptive lookup table. A signal value of 0 applied to the port disables the adaptation, and signal value of 1 enables the adaptation. Setting the signal value to 2 resets the table values to the initial table data.

Adaptive Lookup Table (1D Stair-Fit)

Add cell lock enable/disable port

Selecting this check box creates an additional input port Lock that provides the means for updating only specified cells during a simulation run. A signal value of 0 unlocks the specified cells and signal value of 1 locks the specified cells.

Action for out-of-range input

Ignore or Adapt by extrapolating beyond the extreme breakpoints.

See Also

Adaptive Lookup Table (2D Stair-Fit) | Adaptive Lookup Table (nD Stair-Fit)

Concepts

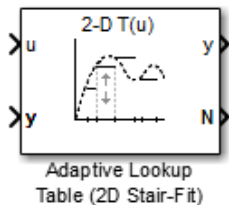
- “What are Adaptive Lookup Tables?”
- “Selecting an Adaptation Method”

Adaptive Lookup Table (2D Stair-Fit)

Purpose Two-dimensional adaptive table lookup

Library Simulink Design Optimization

Description



The Adaptive Lookup Table (2D Stair-Fit) block creates a two-dimensional adaptive lookup table by dynamically updating the underlying lookup table. The block uses the outputs, y , of your system to do the adaptations.

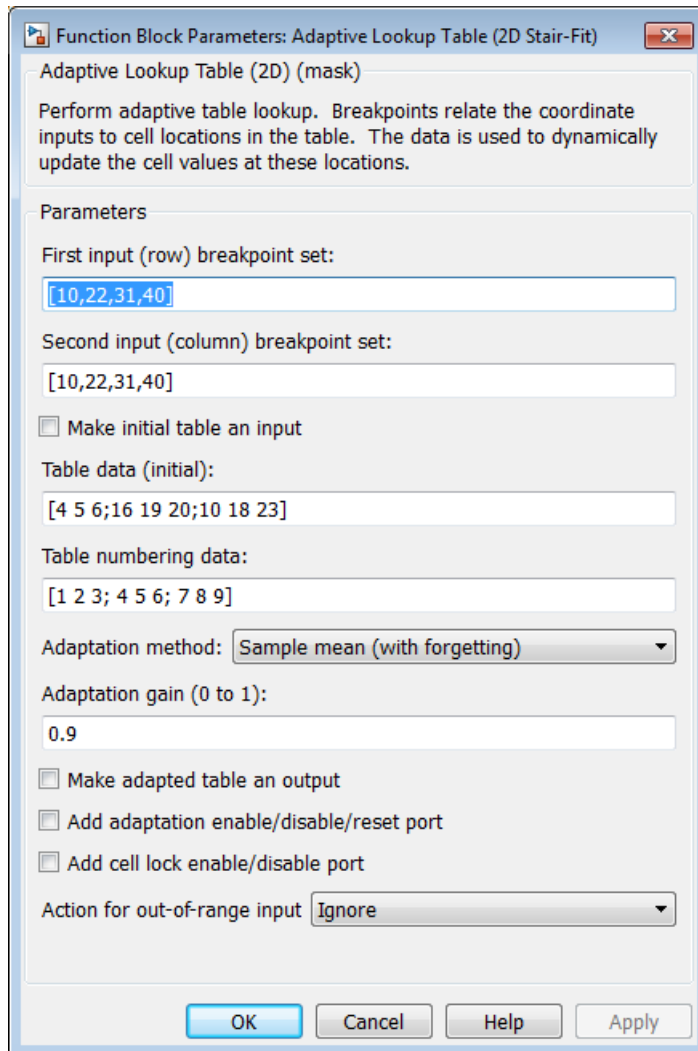
Each indexing parameter u may take a value within a set of adapting data points, which are called *breakpoints*. Two breakpoints in each dimension define a *cell*. The set of all breakpoints in one of the dimensions defines a *grid*. In the two-dimensional case, each cell has four breakpoints and is a flat surface.

You can use the Adaptive Lookup Table (2D Stair-Fit) block to model time-varying systems with two inputs.

Data Type Support

Doubles only

Adaptive Lookup Table (2D Stair-Fit)



Dialog Box

First input (row) breakpoint set

The vector of values containing possible block input values for the first input variable. The first input vector must be monotonically increasing.

Adaptive Lookup Table (2D Stair-Fit)

Second input (column) breakpoint set

The vector of values containing possible block input values for the second input variable. The second input vector must be monotonically increasing.

Make initial table an input

Selecting this check box forces the Adaptive Lookup Table (2D Stair-Fit) block to ignore the **Table data (initial)** parameter, and creates a new input port T_{in} . Use this port to input the table data.

Table data (initial)

The initial table output values. This 2-by-2 matrix must be of size $(n-1)$ -by- $(m-1)$, where n is the number of first input breakpoints and m is the number of second input breakpoints.

Table numbering data

Number values assigned to cells. This matrix must be the same size as the table data matrix, and each value must be unique.

Adaptation method

Choose **Sample mean** or **Sample mean with forgetting**. **Sample mean** averages all the values received within a cell. **Sample mean with forgetting** gives more weight to the new data. How much weight is determined by the **Adaptation gain** parameter. For more information, see “Selecting an Adaptation Method”.

Adaptation gain (0 to 1)

A number from 0 to 1 that regulates the weight given to new data during the adaptation. A 0 means short memory (last data becomes the table value), and 1 means long memory (average all data received in a cell).

Make adapted table an output

Selecting this check box creates an additional output port T_{out} for the adapted table.

Add adaptation enable/disable/reset port

Selecting this check box creates an additional input port **Enable** that enables, disables, or resets the adaptive lookup table. A signal value of 0 applied to the port disables the adaptation, and

Adaptive Lookup Table (2D Stair-Fit)

signal value of 1 enables the adaptation. Setting the signal value to 2 resets the table values to the initial table data.

Add cell lock enable/disable port

Selecting this check box creates an additional input port **Lock** that provides the means for updating only specified cells during a simulation run. A signal value of 0 unlocks the specified cells and signal value of 1 locks the specified cells.

Action for out-of-range input

Ignore or Adapt by extrapolating beyond the extreme breakpoints.

See Also

Adaptive Lookup Table (1D Stair-Fit) | Adaptive Lookup Table (nD Stair-Fit)

Concepts

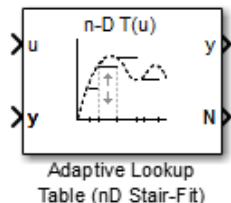
- “What are Adaptive Lookup Tables?”
- “Selecting an Adaptation Method”

Adaptive Lookup Table (nD Stair-Fit)

Purpose Adaptive lookup table of arbitrary dimension

Library Simulink Design Optimization

Description



The Adaptive Lookup Table (nD Stair-Fit) block creates an adaptive lookup table of arbitrary dimension by dynamically updating the underlying lookup table. The block uses the outputs of your system to do the adaptations.

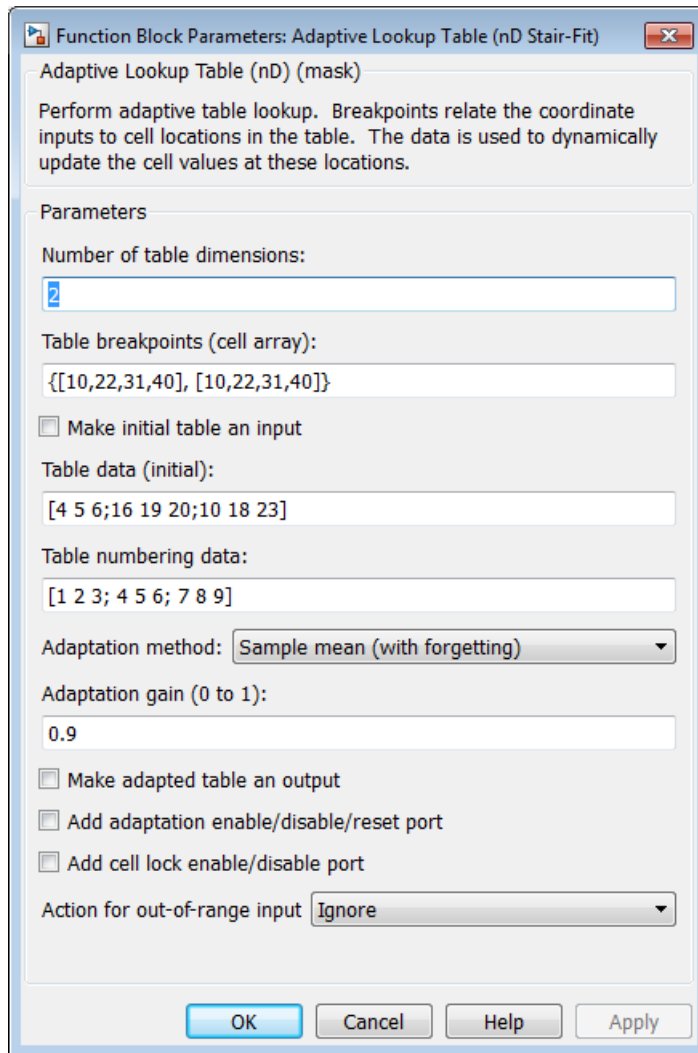
Each indexing parameter may take a value within a set of adapting data points, which are called *breakpoints*. Breakpoints in each dimension define a *cell*. The set of all breakpoints in one of the dimensions defines a *grid*. In the n-dimensional case, each cell has two n breakpoints and is an (n-1) hypersurface.

You can use the Adaptive Lookup Table (nD Stair-Fit) block to model time-varying systems with 2 or more inputs.

Data Type Support

Doubles only

Adaptive Lookup Table (nD Stair-Fit)



Dialog Box

Number of table dimensions

The number of dimensions for the adaptive lookup table.

Adaptive Lookup Table (nD Stair-Fit)

Table breakpoints (cell array)

A set of one-dimensional vectors that contains possible block input values for the input variables. Each input row must be monotonically increasing, but the rows do not have to be the same length. For example, if the **Number of table dimensions** is 3, you can set the table breakpoints as follows:

```
{[1 2 3], [5 7], [1 3 5 7]}
```

Make initial table an input

Selecting this check box forces the Adaptive Lookup Table (nD Stair-Fit) block to ignore the **Table data (initial)** parameter, and creates a new input port T_{in} . Use this port to input the table data.

Table data (initial)

The initial table output values. This (n-D) array must be of size (n-1)-by-(n-1) ... -by- (n-1), (D times), where D is the number of dimensions and n is the number of input breakpoints.

Table numbering data

Number values assigned to cells. This vector must be the same size as the table data array, and each value must be unique.

Adaptation method

Choose **Sample mean** or **Sample mean with forgetting**. **Sample mean** averages all the values received within a cell. **Sample mean with forgetting** gives more weight to the new data. How much weight is determined by the **Adaptation gain** parameter. For more information, see “Selecting an Adaptation Method”.

Adaptation gain (0 to 1)

A number from 0 to 1 that regulates the weight given to new data during the adaptation. A 0 means short memory (last data becomes the table value), and 1 means long memory (average all data received in a cell).

Make adapted table an output

Selecting this check box creates an additional output port T_{out} for the adapted table.

Adaptive Lookup Table (nD Stair-Fit)

Note The Adaptive Lookup Table (n-D Stair Fit) block cannot output a table of 3 or more dimensions.

Add adaptation enable/disable/reset port

Selecting this check box creates an additional input port **Enable** that enables, disables, or resets the adaptive lookup table. A signal value of 0 applied to the port disables the adaptation, and signal value of 1 enables the adaptation. Setting the signal value to 2 resets the table values to the initial table data.

Add cell lock enable/disable port

Selecting this check box creates an additional input port **Lock** that provides the means for updating only specified cells during a simulation run. A signal value of 0 unlocks the specified cells and signal value of 1 locks the specified cells.

Action for out-of-range input

Ignore or Adapt by extrapolating beyond the extreme breakpoints.

See Also

Adaptive Lookup Table (1D Stair-Fit) | Adaptive Lookup Table (2D Stair-Fit)

Related Examples

- “Model Engine Using n-D Adaptive Lookup Table”

Concepts

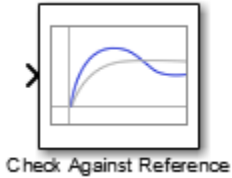
- “What are Adaptive Lookup Tables?”
- “Selecting an Adaptation Method”

Check Against Reference

Purpose Check that model signal tracks reference signal during simulation

Library Simulink Design Optimization

Description Check that a signal remains within tolerance bounds of a reference signal during simulation:



- If all bounds are satisfied, the block does nothing.
- If a bound is not satisfied, the block asserts, and a warning message appears at the MATLAB® prompt. You can also specify that the block:
 - Evaluate a MATLAB expression.
 - Stop the simulation and bring that block into focus.

During simulation, the block can also output a logical assertion signal:

- If all bounds are satisfied, the signal is true (1).
- If a bound is not satisfied, the signal is false (0).

You can add Check Against Reference blocks on multiple signals to check that they track reference signals.

You can also plot the reference signal on a time plot to graphically verify that the signal tracks that signal.

This block and the other blocks in the Model Verification library test that a signal remains within specified time-domain characteristic bounds. When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.

If the signal does not satisfy the bounds, you can optimize the model parameters to satisfy the bounds. If you have Simulink® Control Design™ software, you can add frequency-domain bounds such as Bode magnitude and optimize the model response to satisfy both time- and frequency-domain requirements.

The block can be used in all simulation modes for signal monitoring but only in Normal or Accelerator simulation mode for response optimization.

Parameters

Task	Parameters
Specify a reference signal to: <ul style="list-style-type: none">• Assert that a signal tracks the reference• Optimize model response so that a signal tracks the reference	Include reference signal tracking in assertion in Bounds tab.
Specify assertion options (only when you specify reference to track).	In the Assertion tab: <ul style="list-style-type: none">• Enable assertion• Simulation callback when assertion fails (optional)• Stop simulation when assertion fails• Output assertion signal
Open Design Optimization tool to optimize model response	Click Response Optimization
Plot reference signal	Click Show Plot .
Display plot window instead of Block Parameters dialog box on double-clicking the block.	Show plot on block open

Check Against Reference

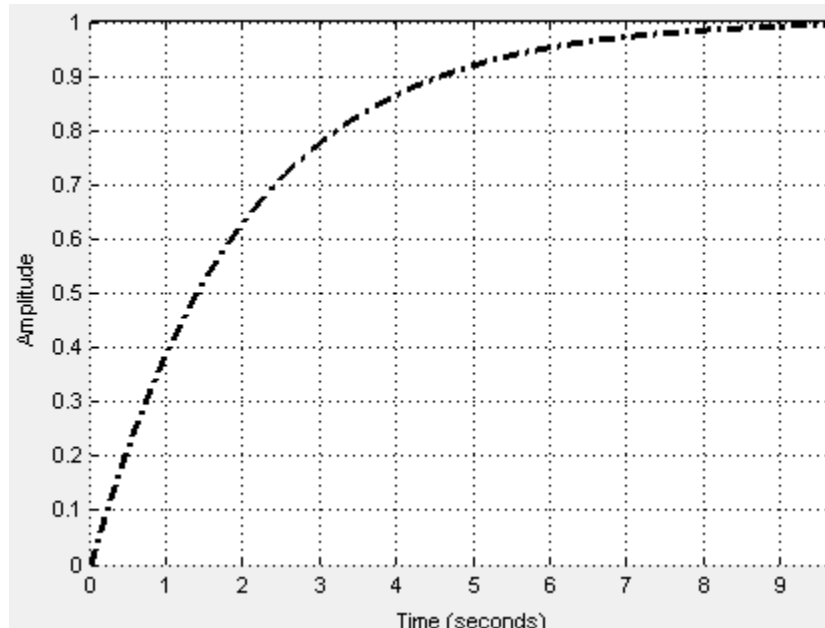
Include reference signal tracking in assertion

Check that the signal does not track the reference signal specified in “Times (seconds)” on page 1-18 and “Amplitudes” on page 1-19 during simulation.

The software displays a warning if the signal does not track the reference signal.

This parameter is used only if **Enable assertion** in the **Assertion** tab is selected.

The reference signal also appears on a time plot if you click **Show Plot**, as shown in the next figure.



If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Settings

Default: On



On

Check that the signal tracks the specified reference signal during simulation.



Off

Do not check that the signal tracks the specified reference signal during simulation.

Tips

- Clearing this parameter disables the reference signal and the software stops checking that the signal tracks the reference during simulation.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: EnableReferenceBound

Type: string

Value: 'on' | 'off'

Default: 'on'

Check Against Reference

Times (seconds)

Time vector for the reference signal. Specify the corresponding amplitudes in “Amplitudes” on page 1-19.

Settings

Default: linspace(0,10)

Command-Line Information

Parameter: ReferenceTimes

Type: string

Value: linspace(0,10) | vector of positive values of the same dimension as the amplitude vector |

Default: linspace(0,10)

Amplitudes

Amplitude of the reference signal corresponding to the time vector specified in “Times (seconds)” on page 1-18.

Settings

Default: `1-exp(-linspace(0,10)/2)`

Command-Line Information

Parameter: `ReferenceAmplitudes`

Type: `string`

Value: `1-exp(-linspace(0,10)/2)` | vector of integers of the same dimension as the time vector

Default: `1-exp(-linspace(0,10)/2)`

Check Against Reference

Absolute tolerance

Absolute tolerance used to determine bounds as the signal approaches the reference signal.

During simulation, the signal must remain within upper and lower limits respective to the reference signal given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

where y_r is the value of the reference at a certain time, y_u and y_l are the upper and lower tolerance bounds corresponding to that time point.

The block asserts if the signal violates these limits.

Settings

Default: $\text{eps}^{(1/3)}$

Minimum: 0

Command-Line Information

Parameter: AbsTolerance

Type: string

Value: $\text{eps}^{(1/3)}$ | positive real scalar

Default: $\text{eps}^{(1/3)}$

Relative tolerance

Relative tolerance used to determine bounds as the signal approaches the reference signal.

During simulation, the signal must remain within upper and lower limits respective to the reference signal given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

where y_r is the value of the reference at a certain time, y_u and y_l are the upper and lower tolerance bounds corresponding to that time point.

The block asserts if the signal violates these limits.

Settings

Default: 0.01

Minimum: 0

Command-Line Information

Parameter: RelTolerance

Type: string

Value: 0.01 | positive real scalar

Default: 0.01

Check Against Reference

Enable assertion

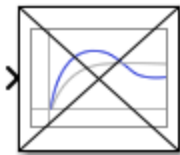
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If the assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)**.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails**.

This parameter has no effect if you do not specify any bounds.

Clearing this parameter disables assertion, i.e., the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



In the Configuration Parameters dialog box of the Simulink model, the **Model Verification block enabling** option in the **Debugging** area of **Data Validity** node, lets you to enable or disable all model verification blocks in a model, regardless of the setting of this option.

Settings

Default: On



Check that bounds included for assertion in the **Bounds** tab are satisfied during simulation. A warning, reporting assertion failure, is displayed at the MATLAB prompt if bounds are violated.

Off

Do not check that bounds included for assertion are satisfied during simulation.

Dependencies

This parameter enables:

- **Simulation callback when assertion fails (optional)**
- **Stop simulation when assertion fails**

Command-Line Information

Parameter: enabled

Type: string

Value: 'on' | 'off'

Default: 'on'

Check Against Reference

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Settings

Default: []

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: callback

Type: string

Value: '' | MATLAB expression

Default: ''

Stop simulation when assertion fails

Stop the simulation when a bound specified in the **Bounds** tab is violated during simulation, i.e., assertion fails.

If you run the simulation from a Simulink model window, the Simulation Diagnostics window opens to display an error message. The block where the bound violation occurs is highlighted in the model.

Settings

Default: Off

On

Stop simulation if a bound specified in the **Bounds** tab is violated.

Off

Continue simulation if a bound is violated and produce a warning message at the MATLAB prompt.

Tips

- Because selecting this option stops the simulation as soon as the assertion fails, assertion failures that might occur later during the simulation are not reported. If you want *all* assertion failures to be reported, do not select this option.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: stopWhenAssertionFail

Type: string

Value: 'on' | 'off'

Default: 'off'

Check Against Reference

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds, i.e., all bounds are satisfied
- False (0) if assertion fails, i.e., a bound is violated.

The output signal data type is Boolean only if the **Implement logic signals as Boolean data** option in the **Optimization** pane of the Configuration Parameters dialog box of the Simulink model is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

Settings

Default:Off



On

Output a Boolean signal to indicate assertion status. Adds a port to the block.



Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink Control Design and Simulink Verification Blocks”.

Command-Line Information

Parameter: export


Type: string

Value: 'on' | 'off'

Default: 'off'

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see **Show Plot**.

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when double-clicking the block.

Command-Line Information

Parameter: LaunchViewOnOpen

Type: string

Value: 'on' | 'off'

Default: 'off'

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

You must click this button before you simulate the model to view the system characteristics or signal.

You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.





- Bounds

Check Against Reference

You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds, see the individual reference pages.

You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking  or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking  or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.
- Simulating the model by clicking  or selecting **Simulation > Start**. This action also linearizes the portion of the model between the specified linearization input and output.
- Adding legend on the linear system characteristic plot by clicking .

A new option **Response Optimization** appears under **Tools** of Simulink Control Design Model Verification blocks if Simulink Design Optimization™ is installed.

Response Optimization

Open the Design Optimization tool to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button appears in Simulink Control Design “Model Verification” Block Parameters dialog box only if you have Simulink Design Optimization software.

See Also

- “Design Optimization to Meet Step Response Requirements (GUI)”
- “Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)”

See Also

- Check Custom Bounds
- Check Step Response Characteristics

Tutorials

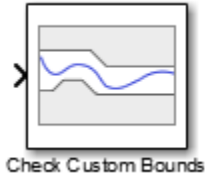
“Design Optimization to Track Reference Signal (GUI)”

Check Custom Bounds

Purpose Check that signal satisfies upper and lower bounds during simulation

Library Simulink Design Optimization

Description Check that a signal satisfies upper and lower bounds during simulation:



- If all bounds are satisfied, the block does nothing.
- If a bound is not satisfied, the block asserts, and a warning message appears at the MATLAB prompt. You can also specify that the block:
 - Evaluate a MATLAB expression.
 - Stop the simulation and bring that block into focus.

During simulation, the block can also output a logical assertion signal:

- If all bounds are satisfied, the signal is true (1).
- If a bound is not satisfied, the signal is false (0).

You can add Check Custom Bounds blocks on multiple signals to check that they satisfy the bounds.

You can also plot the bounds on a time plot to graphically verify that the signal satisfies the bounds.

This block and the other blocks in the Model Verification library test that a signal remains within specified time-domain characteristic bounds. When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.

If the signal does not satisfy the bounds, you can optimize the model parameters to satisfy the bounds. If you have Simulink Control Design software, you can add frequency-domain bounds such as Bode magnitude and optimize the model response to satisfy both time- and frequency-domain requirements.

The block can be used in all simulation modes for signal monitoring but only in Normal or Accelerator simulation mode for response optimization.

Parameters

Task	Parameters
Specify upper and lower bounds to: <ul style="list-style-type: none"> • Assert that a signal satisfies the bounds • Optimize model response so that a signal satisfies the bounds 	In the Bounds tab: <ul style="list-style-type: none"> • Include upper bound in assertion • Include lower bound in assertion
Specify assertion options (only when you specify upper and lower bounds).	In the Assertion tab: <ul style="list-style-type: none"> • Enable assertion • Simulation callback when assertion fails (optional) • Stop simulation when assertion fails • Output assertion signal
Open Design Optimization tool to optimize model response	Click Response Optimization
Plot upper and lower bounds	Click Show Plot .
Display plot window instead of Block Parameters dialog box on double-clicking the block.	Show plot on block open

Check Custom Bounds

Include upper bound in assertion

Check that a signal is less than or equal to upper bounds, specified in **Times (seconds)** and **Amplitudes**, during simulation. The software displays a warning if the signal violates the upper bounds.

This parameter is used for assertion only if **Enable assertion** in the **Assertion** tab is selected.

You can specify multiple upper bounds on various model signals. The bounds also appear on the time plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Settings

Default: On



On

Check that the signal satisfies the specified upper bounds during simulation.

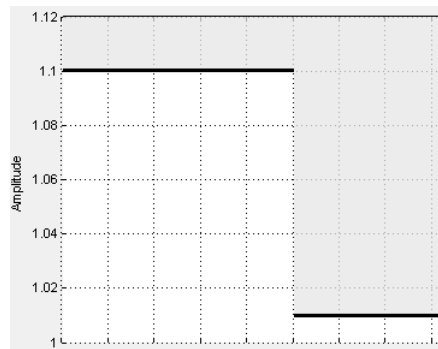


Off

Do not check that the signal satisfies the specified upper bounds during simulation.

Tips

- Clearing this parameter disables the upper bounds and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.



- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: EnableUpperBound

Type: string

Value: 'on' | 'off'

Default: 'on'

Check Custom Bounds

Times (seconds)

Time vector for one or more upper bound segments, specified in seconds.

Specify the corresponding amplitude values in **Amplitudes**.

Settings

Default: [0 5; 5 10]

Must be specified as start and end times:

- Positive finite numbers for a single bound with one edge.
- Matrix of positive finite numbers for a single bound with multiple edges.

For example, type [0.1 1;1 10] for two edges at times [0.1 1] and [1 10].

- Cell array of matrices with positive finite numbers for multiple bounds.

Tips

- To assert that amplitudes that correspond to the time vectors are satisfied, select both **Include upper bound in assertion** and **Enable assertion**.
- You can add or modify start and end times from the plot window:
 - To add new time vectors, right-click the yellow area on the plot, and select **Edit**. Click **Insert** to add a new row to the Edit Bound dialog box. Specify the start and end times of the new bound segment in the **Time** column. Specify the corresponding amplitudes in the **Amplitude** column.
 - To modify the start and end times, drag the bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new times in the **Time** column.

You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: UpperBoundTimes

Type: string

Value: [0 5; 5 10] | positive finite numbers | matrix of positive finite numbers | matrix of positive finite numbers cell array of matrices with positive finite numbers. Must be specified inside single quotes ('').

Default: [0 5; 5 10]

Check Custom Bounds

Amplitudes

Amplitude values for one or more upper bound segments.

Specify the corresponding start and end times in **Times (seconds)**.

Settings

Default: [1.1 1.1; 1.01 1.01]

Must be specified as start and end amplitudes:

- Finite numbers for a single bound with one edge
- Matrix of finite numbers for a single bound with multiple edges.
For example, type [0 1; 2 3] for two edges at amplitudes [0 1] and [2 3].
- Cell array of matrices with finite numbers for multiple bounds

Tips

- To assert that amplitude bounds are satisfied, select both **Include upper bound in assertion** and **Enable assertion**.
- You can add or modify amplitudes from the plot window:
 - To add new amplitudes, right-click the plot, and select **Edit**. Click **Insert** to add a new row to the Edit Bound dialog box. Specify the start and end amplitudes of the new bound segment in the **Amplitude** column. Specify the corresponding start and end times in the **Time** column.
 - To modify the start and end amplitudes, drag the bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new amplitudes in the **Amplitude** column.

You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: UpperBoundAmplitudes

Type: string

Value: [1.1 1.1; 1.01 1.01] | finite numbers | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes ('').

Default: [1.1 1.1; 1.01 1.01]

Check Custom Bounds

Include lower bound in assertion

Check that a signal is greater than or equal to lower bounds, specified in **Times (seconds)** and **Amplitudes**, during simulation.

This parameter is used for assertion only if **Enable assertion** in the **Assertion** tab is selected.

You can specify multiple lower bounds on various model signals. The bounds also appear on the time plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Settings

Default: Off



On

Check that the signal satisfies the specified lower bounds during simulation.

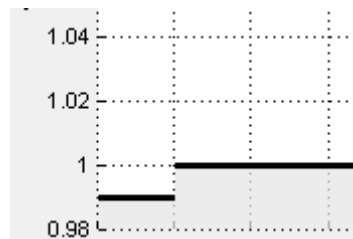


Off

Do not check that the signal satisfies the specified lower bounds during simulation.

Tips

- Clearing this parameter disables the lower bounds and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.



- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: EnableLowerBound

Type: string
Value: 'on' | 'off'
Default: 'off'

Check Custom Bounds

Times (seconds)

Time vector for one or more lower bound segments, specified in seconds.

Specify the corresponding amplitude values in **Amplitudes**

Settings

Default: []

Must be specified as start and end times:

- Positive finite numbers for a single bound with one edge.
- Matrix of positive finite numbers for a single bound with multiple edges.

For example, type [0.1 1;1 10] for two edges at times [0.1 1] and [1 10].

- Cell array of matrices with positive finite numbers for multiple bounds.

Tips

- To assert that amplitudes that correspond to the time vectors are satisfied, select both **Include lower bound in assertion** and **Enable assertion**.
- You can add or modify start and end times from the plot window:
 - To add new time vectors, right-click the yellow area on the plot, and select **Edit**. Click **Insert** to add a new row to the Edit Bound dialog box. Specify the start and end times of the new bound segment in the **Time** column. Specify the corresponding amplitudes in the **Amplitude** column.
 - To modify the start and end times, drag the bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new times in the **Time** column.

You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: LowerBoundTimes

Type: string

Value: [] | positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (' ').

Default: []

Check Custom Bounds

Amplitudes

Amplitude values for one or more lower bound segments.

Specify the corresponding start and end times in **Times (seconds)**.

Settings

Default: []

Must be specified as start and end amplitudes:

- Finite numbers for a single bound with one edge
- Matrix of finite numbers for a single bound with multiple edges.

For example, type [0 1; 2 3] for two edges at amplitudes [0 1] and [2 3].

- Cell array of matrices with finite numbers for multiple bounds

Tips

- To assert that amplitude bounds are satisfied, select both **Include lower bound in assertion** and **Enable assertion**.
- You can add or modify amplitudes from the plot window:
 - To add new amplitudes, right-click the plot, and select **Edit**. Click **Insert** to add a new row to the Edit Bound dialog box. Specify the start and end amplitudes of the new bound segment in the **Amplitude** column. Specify the corresponding start and end times in the **Time** column.
 - To modify the start and end amplitudes, drag the bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new amplitudes in the **Amplitude** column.

You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: LowerBoundAmplitudes

Type: string

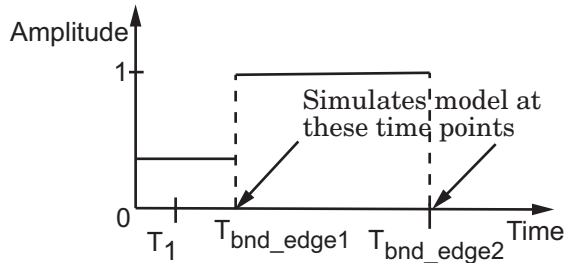
Value: [] | finite numbers | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (' ').

Default: []

Check Custom Bounds

Enable zero-crossing detection

Ensure that the software simulates the model to produce output at the bound edges. Simulating the model at the bound edges prevents the simulation solver from missing a bound edge without asserting that the signal satisfies that bound.



For more information on zero-crossing detection, see “Zero-Crossing Detection” in the *Simulink User Guide*.

Settings

Default: On



On

Simulate model at the bound edges

This setting is ignored if the Simulink solver is fixed step.



Off

Do not simulate model at the bound edges. The software may not compute the output at the bound edges.

Command-Line Information

Parameter: ZeroCross

Type: string

Value: 'on' | 'off'

Default: 'on'

Enable assertion

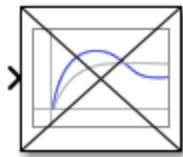
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If the assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)**.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails**.

This parameter has no effect if you do not specify any bounds.

Clearing this parameter disables assertion, i.e., the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



In the Configuration Parameters dialog box of the Simulink model, the **Model Verification block enabling** option in the **Debugging** area of **Data Validity** node, lets you to enable or disable all model verification blocks in a model, regardless of the setting of this option.

Settings

Default: On

On

Check that bounds included for assertion in the **Bounds** tab are satisfied during simulation. A warning, reporting assertion failure, is displayed at the MATLAB prompt if bounds are violated.

Check Custom Bounds



Off

Do not check that bounds included for assertion are satisfied during simulation.

Dependencies

This parameter enables:

- **Simulation callback when assertion fails (optional)**
- **Stop simulation when assertion fails**

Command-Line Information

Parameter: enabled

Type: string

Value: 'on' | 'off'

Default: 'on'

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Settings

Default: []

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: callback

Type: string

Value: ' ' | MATLAB expression

Default: ' '

Check Custom Bounds

Stop simulation when assertion fails

Stop the simulation when a bound specified in the **Bounds** tab is violated during simulation, i.e., assertion fails.

If you run the simulation from a Simulink model window, the Simulation Diagnostics window opens to display an error message. The block where the bound violation occurs is highlighted in the model.

Settings

Default: Off



On

Stop simulation if a bound specified in the **Bounds** tab is violated.



Off

Continue simulation if a bound is violated and produce a warning message at the MATLAB prompt.

Tips

- Because selecting this option stops the simulation as soon as the assertion fails, assertion failures that might occur later during the simulation are not reported. If you want *all* assertion failures to be reported, do not select this option.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: stopWhenAssertionFail

Type: string

Value: 'on' | 'off'

Default: 'off'

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds, i.e., all bounds are satisfied
- False (0) if assertion fails, i.e., a bound is violated.

The output signal data type is Boolean only if the **Implement logic signals as Boolean data** option in the **Optimization** pane of the Configuration Parameters dialog box of the Simulink model is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

Settings

Default: Off

On

Output a Boolean signal to indicate assertion status. Adds a port to the block.

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink Control Design and Simulink Verification Blocks”.

Command-Line Information

Parameter: export

Type: string


Value: 'on' | 'off'

Default: 'off'

Check Custom Bounds

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see **Show Plot**.

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when double-clicking the block.

Command-Line Information

Parameter: LaunchViewOnOpen

Type: string

Value: 'on' | 'off'

Default: 'off'

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

You must click this button before you simulate the model to view the system characteristics or signal.





You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.

- Bounds

You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds, see the individual reference pages.

You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking  or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking  or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.
- Simulating the model by clicking  or selecting **Simulation > Start**. This action also linearizes the portion of the model between the specified linearization input and output.
- Adding legend on the linear system characteristic plot by clicking .

A new option **Response Optimization** appears under **Tools** of Simulink Control Design Model Verification blocks if Simulink Design Optimization is installed.

Response Optimization

Open the Design Optimization tool to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button appears in Simulink Control Design “Model Verification” Block Parameters dialog box only if you have Simulink Design Optimization software.

Check Custom Bounds

See Also

- “Design Optimization to Meet Step Response Requirements (GUI)”
- “Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)”

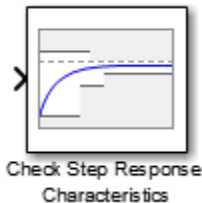
Purpose

Check that model signal satisfies step response bounds during simulation

Library

Simulink Design Optimization

Description



Check that a signal satisfies step response bounds during simulation:

- If all bounds are satisfied, the block does nothing.
- If a bound is not satisfied, the block asserts, and a warning message appears at the MATLAB prompt. You can also specify that the block:
 - Evaluate a MATLAB expression.
 - Stop the simulation and bring that block into focus.

During simulation, the block can also output a logical assertion signal:

- If all bounds are satisfied, the signal is true (1).
- If a bound is not satisfied, the signal is false (0).

You can add Check Step Response Characteristics blocks on multiple signals to check that they satisfy the bounds.

You can also plot the bounds on a time plot to graphically verify that the signal satisfies the bounds.

This block and the other blocks in the Model Verification library test that a signal remains within specified time-domain characteristic bounds. When a model does not violate any bound, you can disable the

Check Step Response Characteristics

block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.

If the signal does not satisfy the bounds, you can optimize the model parameters to satisfy the bounds. If you have Simulink Control Design software, you can add frequency-domain bounds such as Bode magnitude and optimize the model response to satisfy both time- and frequency-domain requirements.

The block can be used in all simulation modes for signal monitoring but only in Normal or Accelerator simulation mode for response optimization.

Parameters

Task	Parameters
Specify step response bounds to: <ul style="list-style-type: none">• Assert that a signal satisfies the bounds• Optimize model response so that a signal satisfies the bounds	Include step response bound in assertion in Bounds tab.
Specify assertion options (only when you specify step response bounds).	In the Assertion tab: <ul style="list-style-type: none">• Enable assertion• Simulation callback when assertion fails (optional)• Stop simulation when assertion fails• Output assertion signal
Open Design Optimization tool to optimize model response	Click Response Optimization

Check Step Response Characteristics

Task	Parameters
Plot step response	Click Show Plot .
Display plot window instead of Block Parameters dialog box on double-clicking the block.	Show plot on block open

Check Step Response Characteristics

Include step response bound in assertion

Check that the step response satisfies *all* the characteristics specified in:

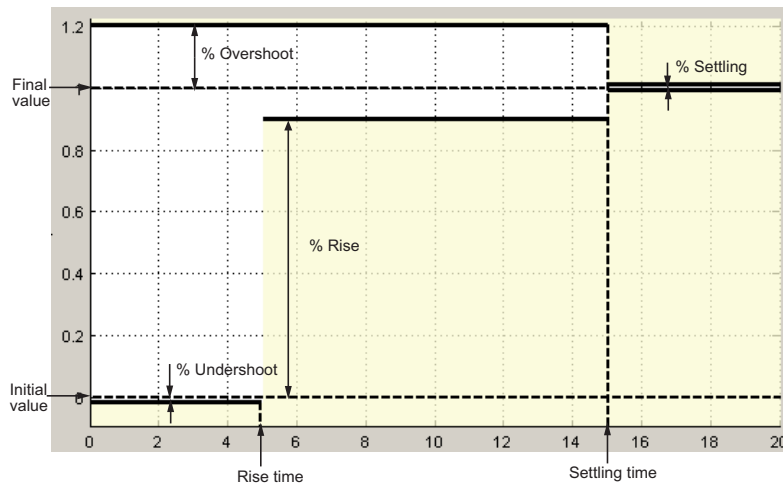
- **Step time (seconds)**
- **Initial value**
- **Final Value**
- **Rise time (seconds)**
- **% Rise**
- **Settling time (seconds)**
- **% Settling**
- **% Overshoot**
- **% Undershoot**

The software displays a warning if the signal violates the specified step response characteristics.

This parameter is used for assertion only if **Enable assertion** in the **Assertion** tab is selected.

The bounds also appear on the step response plot if you click **Show Plot**, as shown in the next figure.

Check Step Response Characteristics



By default, the line segments represent the following step response requirements:

- Amplitude less than or equal to -0.01 up to the rise time of 5 seconds for 1% undershoot
- Amplitude between 0.9 and 1.2 up to the settling time of 15 seconds
- Amplitude equal to 1.2 for 20% overshoot up to the settling time of 15 seconds
- Amplitude between 0.99 and 1.01 beyond the settling time for 2% settling

If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Settings

Default: On



On

Check that the step response satisfies the specified bounds during simulation.

Check Step Response Characteristics

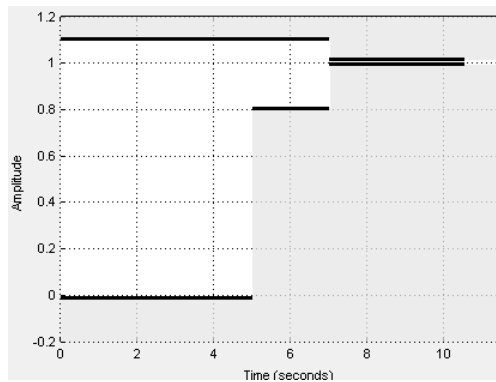


Off

Do not check that the step response satisfies the specified bounds during simulation.

Tips

- Clearing this parameter disables the step response bounds and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.



- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: EnableStepResponseBound

Type: string

Value: 'on' | 'off'

Default: 'on'

Check Step Response Characteristics

Step time (seconds)

Time, in seconds, when the step response starts.

Settings

Default: 0

Minimum: 0

Finite real nonnegative scalar.

Tips

- To assert that step time value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the step time value from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **Step time**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: StepTime

Type: string

Value: 0 | finite real nonnegative scalar. Must be specified inside single quotes (' ').

Default: 0

Check Step Response Characteristics

Initial value

Value of the signal level before the step response starts.

Settings

Default: 0

Finite real scalar not equal to the final value.

Tips

- To assert that initial value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the initial value from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **Initial value**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: InitialValue

Type: string

Value: 0 | finite real scalar not equal to final value.

Must be specified inside single quotes (' ').

Default: 0

Check Step Response Characteristics

Final value

Final value of the step response.

Settings

Default: 1

Finite real scalar not equal to the initial value.

Tips

- To assert that final value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the final value from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **Final value**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: FinalValue

Type: string

Value: 1 | finite real scalar not equal to the initial value. Must be specified inside single quotes (' ').

Default: 1

Check Step Response Characteristics

Rise time (seconds)

Time taken, in seconds, for the signal to reach a percentage of the final value specified in % **Rise**.

Settings

Default: 5

Minimum: 0

Finite positive real scalar, less than the settling time.

Tips

- To assert that rise time value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the rise time from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **Rise time**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: RiseTime

Type: string

Value: 5 | finite positive real scalar. Must be specified inside single quotes ('').

Default: 5

Check Step Response Characteristics

% Rise

The percentage of final value used with the **Rise time** to define the overall rise time characteristics.

Settings

Default: 80

Minimum: 0

Maximum: 100

Positive real scalar, less than (100 – % settling).

Tips

- To assert that percent rise value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the percent rise from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **% Rise**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: PercentRise

Type: string

Value: 80 | positive scalar less than (100 – % settling).
Must be specified inside single quotes (' ').

Default: 80

Check Step Response Characteristics

Settling time (seconds)

The time, in seconds, taken for the signal to settle within a specified range around the final value. This settling range is defined as the final value plus or minus the percentage of the final value, specified in % **Settling**.

Settings

Default: 7

Finite positive real scalar, greater than rise time.

Tips

- To assert that final value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the settling time from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **Settling time**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: SettlingTime

Type: string

Value: 7 | positive finite real scalar greater than rise time. Must be specified inside single quotes (' ').

Default: 7

% Settling

The percentage of the final value that defines the settling range of the **Settling time** characteristic.

Settings

Default: 1

Minimum: 0

Maximum: 100

Real positive finite scalar, less than (100 – % rise) and less than % overshoot.

Tips

- To assert that percent settling value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the percent settling from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **% Settling**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: PercentSettling

Type: string

Value: 1 | Real positive finite scalar less than (100 % rise) and less than % overshoot. Must be specified inside single quotes (' ').

Default: 1

Check Step Response Characteristics

% Overshoot

The amount by which the signal can exceed the final value before settling, specified as a percentage.

Settings

Default: 10

Minimum: 0

Maximum: 100

Positive real scalar, greater than % settling.

Tips

- To assert that percent overshoot value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the percent overshoot from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **% Overshoot**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: PercentOvershoot

Type: string

Value: 10 | Positive real scalar greater than % settling. Must be specified inside single quotes (' ').

Default: 10

Check Step Response Characteristics

% Undershoot:

The amount by which the signal can undershoot the initial value, specified as a percentage.

Settings

Default: 1

Minimum: 0

Maximum: 100

Positive finite real scalar.

Tips

- To assert that percent undershoot value is satisfied, select both **Include step response bound in assertion** and **Enable assertion**.
- To modify the percent undershoot from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Edit**. Specify the new value in **% Undershoot**. You must click **Update Block** before simulating the model.

Command-Line Information

Parameter: PercentUndershoot

Type: string

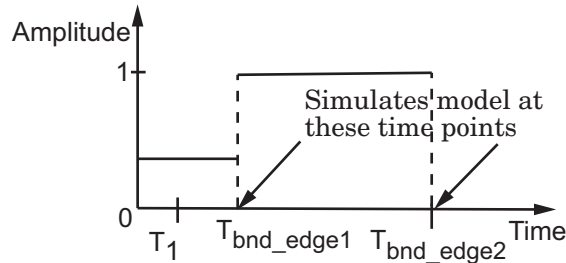
Value: 1 | positive finite real scalar between 0 and 100. Must be specified inside single quotes (' ').

Default: 1

Check Step Response Characteristics

Enable zero-crossing detection

Ensure that the software simulates the model to produce output at the bound edges. Simulating the model at the bound edges prevents the simulation solver from missing a bound edge without asserting that the signal satisfies that bound.



For more information on zero-crossing detection, see “Zero-Crossing Detection” in the *Simulink User Guide*.

Settings

Default: On



On

Simulate model at the bound edges

This setting is ignored if the Simulink solver is fixed step.



Off

Do not simulate model at the bound edges. The software may not compute the output at the bound edges.

Command-Line Information

Parameter: ZeroCross

Type: string

Value: 'on' | 'off'

Default: 'on'

Check Step Response Characteristics

Enable assertion

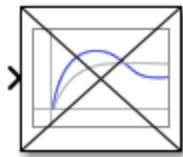
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If the assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)**.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails**.

This parameter has no effect if you do not specify any bounds.

Clearing this parameter disables assertion, i.e., the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



In the Configuration Parameters dialog box of the Simulink model, the **Model Verification block enabling** option in the **Debugging** area of **Data Validity** node, lets you to enable or disable all model verification blocks in a model, regardless of the setting of this option.

Settings

Default: On

On

Check that bounds included for assertion in the **Bounds** tab are satisfied during simulation. A warning, reporting assertion failure, is displayed at the MATLAB prompt if bounds are violated.

Check Step Response Characteristics



Off

Do not check that bounds included for assertion are satisfied during simulation.

Dependencies

This parameter enables:

- **Simulation callback when assertion fails (optional)**
- **Stop simulation when assertion fails**

Command-Line Information

Parameter: enabled

Type: string

Value: 'on' | 'off'

Default: 'on'

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Settings

Default: []

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: callback

Type: string

Value: '' | MATLAB expression

Default: ''

Stop simulation when assertion fails

Stop the simulation when a bound specified in the **Bounds** tab is violated during simulation, i.e., assertion fails.

If you run the simulation from a Simulink model window, the Simulation Diagnostics window opens to display an error message. The block where the bound violation occurs is highlighted in the model.

Settings

Default: Off

On

Stop simulation if a bound specified in the **Bounds** tab is violated.

Off

Continue simulation if a bound is violated and produce a warning message at the MATLAB prompt.

Tips

- Because selecting this option stops the simulation as soon as the assertion fails, assertion failures that might occur later during the simulation are not reported. If you want *all* assertion failures to be reported, do not select this option.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: stopWhenAssertionFail

Type: string

Value: 'on' | 'off'

Default: 'off'

Check Step Response Characteristics

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds, i.e., all bounds are satisfied
- False (0) if assertion fails, i.e., a bound is violated.

The output signal data type is Boolean only if the **Implement logic signals as Boolean data** option in the **Optimization** pane of the Configuration Parameters dialog box of the Simulink model is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

Settings

Default:Off



On

Output a Boolean signal to indicate assertion status. Adds a port to the block.



Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink Control Design and Simulink Verification Blocks”.

Command-Line Information

Parameter: export


Type: string

Value: 'on' | 'off'

Default: 'off'

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see **Show Plot**.

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when double-clicking the block.

Command-Line Information

Parameter: LaunchViewOnOpen

Type: string

Value: 'on' | 'off'

Default: 'off'

Check Step Response Characteristics

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

You must click this button before you simulate the model to view the system characteristics or signal.





You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.

- Bounds

You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds, see the individual reference pages.

You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking  or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking  or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.
- Simulating the model by clicking  or selecting **Simulation > Start**. This action also linearizes the portion of the model between the specified linearization input and output.
- Adding legend on the linear system characteristic plot by clicking .

A new option **Response Optimization** appears under **Tools** of Simulink Control Design Model Verification blocks if Simulink Design Optimization is installed.

Response Optimization

Open the Design Optimization tool to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button appears in Simulink Control Design “Model Verification” Block Parameters dialog box only if you have Simulink Design Optimization software.

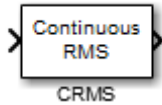
See Also

- “Design Optimization to Meet Step Response Requirements (GUI)”
- “Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)”

Purpose Compute continuous-time, cumulative root mean square (CRMS) of signal

Library Simulink Design Optimization

Description Attach the CRMS block to a signal to compute its continuous-time, cumulative root mean square value. Use in conjunction with the Signal Constraint block to optimize the signal energy.



The continuous-time, cumulative root mean square value of a signal $u(t)$ is defined as

$$R.M.S = \sqrt{\frac{1}{T} \int_0^T \|u(t)\|^2 dt}$$

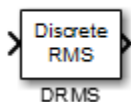
The R.M.S value gives a measure of the average energy in the signal.

See Also DRMS, Signal Constraint

Purpose Compute discrete-time, cumulative root mean square (DRMS) of signal

Library Simulink Design Optimization

Description



Attach the DRMS block to a signal to compute its discrete-time, cumulative root mean square value. Use in conjunction with the Signal Constraint block to optimize the signal energy.

The discrete-time, cumulative root mean square value of a signal $u(t_i)$ is defined as

$$R.M.S = \sqrt{\frac{1}{N} \sum_{i=1}^N \|u(t_i)\|^2}$$

The R.M.S value gives a measure of the average energy in the signal.

See Also CRMS, Signal Constraint

Signal Constraint

Purpose Specify desired signal response

Note Signal Constraint has been removed. Use `sdupdate` to replace it with the equivalent block from the **Signal Constraints** block library.

Library Simulink Design Optimization

Class Reference

param.Continuous

Purpose	Continuous parameter
Syntax	<pre>p = param.Continuous(paramname) p = param.Continuous(paramname,paramvalue)</pre>
Description	<p>A continuous parameter is a numeric parameter that can take any value in a specified interval. The parameter can be scalar- or matrix-valued.</p> <p>Typically, you use continuous parameters to create parametric models and to estimate or optimize tunable parameters in such models.</p>
Construction	<p><code>p = param.Continuous(paramname)</code> constructs a <code>param.Continuous</code> object and assigns the specified parameter name to the <code>Name</code> property and default values to the remaining properties.</p> <p><code>p = param.Continuous(paramname,paramvalue)</code> assigns the specified parameter value to the <code>Value</code> property.</p> <p><code>sdo.getParameterFromModel</code> also constructs a <code>param.Continuous</code> object or an array of <code>param.Continuous</code> objects for Simulink model parameters.</p>
Input Arguments	<p>paramname Parameter name, specified as a string inside single quotes (' ').</p> <p>paramvalue Scalar or matrix numeric double</p>
Properties	<p>Free Flag specifying whether the parameter is tunable or not.</p> <p>Set the <code>Free</code> property to <code>true</code> (1) for tunable parameters and <code>false</code> (0) for parameters you do not want to tune (fixed).</p> <p>The dimension of this property must match the dimension of the <code>Value</code> property.</p> <p>For matrix-valued parameters, you can:</p>

- Fix individual matrix elements. For example `p.Free = [true false; false true]` or `p.Free([2 3]) = false`.
- Use scalar expansion to fix all matrix elements. For example `p.Free = false`.

Default: `true (1)`

Info

Structure array specifying parameter units and labels.

The structure has `Label` and `Unit` fields.

The array dimension must match the dimension of the `Value` property.

Use this property to store parameter units and labels that describe the parameter. For example `p.Info(1,1).Unit = 'N/m'`; or `p.Info(1,1).Label = 'spring constant'`.

Default: `' '` for both `Label` and `Unit` fields

Maximum

Upper bound for the parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued parameters, you can:

- Specify upper bounds on individual matrix elements. For example `p.Maximum([1 4]) = 5`.
- Use scalar expansion to set the upper bound for all matrix elements. For example `p.Maximum = 5`.

Default: `Inf`

Minimum

param.Continuous

Lower bound for the parameter value.

The dimension of this property must match the dimension of the Value property.

For matrix-valued parameters, you can:

- Specify lower bounds on individual matrix elements. For example `p.Minimum([1 4]) = -5`.
- Use scalar expansion to set the lower bound for all matrix elements. For example `p.Minimum = -5`.

Default: `Inf`

Name

Parameter name.

This property is read-only and is set at object construction.

Default: `''`

Scale

Scaling factor used to normalize the parameter value.

The dimension of this property must match the dimension of the Value property.

For matrix-valued parameters, you can:

- Specify scaling for individual matrix elements. For example `p.Scale([1 4]) = 1`.
- Use scalar expansion to set the scaling for all matrix elements. For example `p.Scale = 1`.

Default: `1`

Value

Scalar or matrix value of a parameter.

The dimension of this property is set at object construction.

Default: 0

Methods

isreal

Determine if parameter value, minimum and maximum are real

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a param.Continuous object and specify the maximum value.

```
p = param.Continuous('K',eye(2));  
p.Maximum = 5;
```

Alternatives

“Design Optimization to Meet Step Response Requirements (GUI)”

See Also

sdo.optimize | sdo.getParameterFromModel

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Design Optimization to Meet a Custom Objective (Code)”

How To

- Class Attributes
- Property Attributes

param.State

Superclasses Continuous

Purpose Specify tuning parameters for model states

Description A *state parameter* is a numeric parameter, representing a state associated with a model, that can take any value in a specified interval. The parameter can take scalar or matrix values.

You use state parameters to estimate or specify the initial state values of a model.

Construction You obtain a state parameter using the `sdo.getStateFromModel` function.

For example, use

```
s = sdo.getStateFromModel('sdoMassSpringDamper','Position');
```

to obtain the state parameter of the `Position` block of the `sdoMassSpringDamper` Simulink model.

Properties **Free**

Flag specifying whether the state parameter is tunable or not.

Set the `Free` property to `true` (1) for tunable state parameters and `false` (0) for state parameters you do not want to tune, to designate them as fixed.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued state parameters, you can:

- Fix individual matrix elements. For example, `p.Free = [true false; false true]` or `p.Free([2 3]) = false`.
- Use scalar expansion to fix all matrix elements. For example, `p.Free = false`.

Default: true (1)

Info

Structure array specifying state parameter units and labels.

The structure has `Label` and `Unit` fields.

The array dimension must match the dimension of the `Value` property.

Use this property to store state parameter units and labels. For example, `p.Info(1,1).Unit = 'N/m'`; or `p.Info(1,1).Label = 'spring constant'`.

Default: '' for both `Label` and `Unit` fields

Maximum

Upper bound for the state parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued state parameters, you can:

- Specify upper bounds on individual matrix elements. For example, `p.Maximum([1 4]) = 5`.
- Use scalar expansion to set the upper bound for all matrix elements. For example `p.Maximum = 5`.

Default: Inf

Minimum

Lower bound for the state parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued state parameters, you can:

param.State

- Specify lower bounds on individual matrix elements. For example `p.Minimum([1 4]) = -5`.
- Use scalar expansion to set the lower bound for all matrix elements. For example `p.Minimum = -5`.

Default: `Inf`

Name

State parameter name.

This read-only property is set at object construction.

Default: `''`

Scale

Scaling factor used to normalize the state parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued state parameters, you can:

- Specify scaling for individual matrix elements. For example `p.Scale([1 4]) = 1`.
- Use scalar expansion to set the scaling for all matrix elements. For example `p.Scale = 1`.

Default: `1`

Value

State parameter value.

You can specify the value as either a scalar or a matrix.

The dimension of this property is set at object construction.

Default: `0`

dxFree

Flag specifying whether the state parameter derivative (with respect to time) is tunable or not.

Set the `dxFree` property to `true` (1) for tunable state parameter derivatives and `false` (0) for state parameter derivatives you do not want to tune (fixed).

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued state parameter derivatives, you can:

- Fix individual matrix elements. For example `p.dxFree = [true false; false true]` or `p.dxFree([2 3]) = false`.
- Use scalar expansion to fix all matrix elements. For example `p.dxFree = false`.

Default: `true` (1)

dxValue

State parameter derivative (with respect to time) value.

The dimension of this property must match the dimension of the `Value` property.

Default: 0

Methods

Inherited Methods

`isreal`

Determine if parameter value, minimum and maximum are real

Copy Semantics

`Value`. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Get State Parameters from Model

```
modelname = 'sdoAircraft';  
open_system(modelname);  
  
blockpath = {'sdoAircraft/Actuator Model', ...  
            'sdoAircraft/Controller/Proportional plus integral compensator'};  
  
s = sdo.getStateFromModel(modelname,blockpath);
```

Alternatives

“Specify Known Initial States”

See Also

[sdo.getStateFromModel](#) | [sdo.Experiment](#)

Related Examples

- “Estimate Model Parameter Values (Code)”
- “Estimate Model Parameters and Initial States (Code)”

Concepts

- Class Attributes
- Property Attributes

Purpose	Analysis options for <code>sdo.analyze</code>
Syntax	<pre>opt = sdo.AnalyzeOptions opt = sdo.AnalyzeOptions('Method',method_name)</pre>
Description	Specify analysis method and method options for sensitivity analysis using <code>sdo.analyze</code> .
Construction	<p><code>opt = sdo.AnalyzeOptions</code> creates an <code>sdo.AnalyzeOptions</code> object and assigns default values to the properties.</p> <p>To change a property value, use dot notation. For example:</p> <pre>opt = sdo.AnalyzeOptions; opt.Method = 'StandardizedRegression'; opt.MethodOptions = 'Ranked';</pre> <p><code>opt = sdo.AnalyzeOptions('Method',method_name)</code> sets the value of the <code>Method</code> property to <code>method_name</code>.</p>
Input Arguments	
method_name	<p>Method name, specified as one of the following strings: 'Correlation', 'PartialCorrelation', 'StandardizedRegression', or 'All',</p> <p>To use multiple methods, specify <code>method_name</code> as a cell array of strings.</p> <p>For example, <code>method_name = 'PartialCorrelation'</code>.</p> <p>For information about each method, see the <code>Method</code> property description.</p>

sdo.AnalyzeOptions

Properties

Method

Analysis method used by `sdo.analyze`, specified as one of the following strings or a cell array containing a subset of the following strings:

- 'Correlation' — Calculates the correlation coefficients, R . Use to analyze how a model parameter and the cost function outputs are correlated.

R is calculated as follows:

$$R(i, j) = \frac{C(i, j)}{\sqrt{C(i, i)C(j, j)}}$$
$$C = \text{cov}(x, y)$$
$$= E[(x - \mu_x)(y - \mu_y)]$$
$$\mu_x = E[x]$$
$$\mu_y = E[y]$$

x and y are the input arguments of `sdo.analyze`.

R values are in the [-1 1] range. The (i, j) entry of R indicates the correlation between $x(i)$ and $y(j)$.

- $R(i, j) > 0$ — Variables have positive correlation. The variables increase together.
 - $R(i, j) = 0$ — Variables have no correlation.
 - $R(i, j) < 0$ — Variables have negative correlation. As one variable increases, the other decreases.
- 'PartialCorrelation' (Requires a Statistics Toolbox™ license) — Calculates the partial correlation coefficients, R . Use to analyze how a model parameter and the cost function are correlated, adjusting to remove the effect of the other parameters.

R is calculated using `partialcorri` in the Statistics Toolbox software.

- 'StandardizedRegression' — Calculates the standardized regression coefficients, R . Use when you expect that the model parameters linearly influence the cost function.

R is calculated as follows:

$$R = b_x \frac{\sigma_x}{\sigma_y}$$

Consider a single sample (x_1, \dots, x_{Np}) and the corresponding single output, y . b_x is the regression coefficient vector calculated

using least squares assuming a linear model $\hat{y} = b_0 + \sum_{i=1}^{Np} \hat{b}_{x_i} x_i$. R standardizes each element of b_x by multiplying it with the ratio of the standard deviation of the corresponding x sample (σ_x) to the standard deviation of y (σ_y).

- 'All' — The software calculates results for all applicable combinations of `Method` and `MethodOptions`. This option may be time consuming if you have a large sample set with many parameters and many different cost/constraint outputs.

For x (Ns -by- Np) and y (Ns -by- Nc), all the methods calculate R as an Np -by- Nc table. Here Ns is the number of samples, Np is the number of model parameters, and Nc is the number of cost/constraint function evaluations.

Default: 'Correlation'

MethodOptions

String specifying the analysis method option that `sdo.analyze` uses, specified as one of the following:

- 'Linear' — Pearson analysis.

sdo.AnalyzeOptions

Applicable for all methods.

- 'Ranked' — Ranked transformation or Spearman analysis.

Applicable for all methods.

- 'Kendall' — Kendall's tau.

Applicable when Method is specified as 'Correlation'.

- 'AllApplicable' — Computes each applicable combination of Method and MethodOptions.

Applicable when Method is specified as 'All'.

For more information about these options, see “Sensitivity Analysis Methods”.

Default: 'Linear'

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify Analysis Options

```
opt = sdo.AnalyzeOptions;  
opt.Method = 'PartialCorrelation';  
opt.MethodOptions = 'Ranked';
```

See Also

sdo.analyze

Concepts

- Class Attributes
- Property Attributes
- “Sensitivity Analysis Methods”

Purpose	Cost function evaluation options for <code>sdo.evaluate</code>
Description	Specify options such as evaluation error handling, display settings, and whether to use parallel computing for cost function evaluations using <code>sdo.evaluate</code> .
Construction	<p><code>opt = sdo.EvaluateOptions</code> creates an <code>sdo.EvaluateOptions</code> object and assigns default values to the properties.</p> <p>Use dot notation to modify the property values. For example:</p> <pre>opt = sdo.EvaluateOptions; opt.Display = 'iter';</pre>
Properties	<p>UseParallel</p> <p>Parallel computing option for <code>sdo.evaluate</code>:</p> <ul style="list-style-type: none">• 'never' — Do not use parallel computing during cost function evaluation• 'always' — Use parallel computing during cost function evaluation <p>It is recommended that you also specify values for the <code>EvaluatedModel</code> and <code>ParallelPathDependencies</code> (if needed) properties.</p> <p>Parallel Computing Toolbox™ software must be installed to enable parallel computing for the cost function evaluation.</p> <p>Default: 'never'</p> <p>StopOnEvaluateError</p> <p>Flag to stop <code>sdo.evaluate</code> when a cost function evaluation results in an error, specified as one of the following strings:</p>

sdo.EvaluateOptions

- 'on' — `sdo.evaluate` stops when a cost function evaluation results in an error.
- 'off' — `sdo.evaluate` continues when a cost function evaluation results in an error. `sdo.evaluate` returns the error using the `info` output argument.

Default: 'off'

Display

Level of display messages for cost function evaluations, specified as one of the following strings:

- 'off' — Displays no output
- 'final' — Displays only the final output
- 'iter' — Displays the output for each evaluation

Default: 'final'

ParallelPathDependencies

Path dependencies to use during parallel evaluation, specified as a cell array of strings.

These path dependencies are temporarily added to the workers during parallel evaluation.

Default: {}

EvaluatedModel

Name of Simulink model name to be evaluated, specified as a string.

This property is used to configure the model for parallel evaluation (`UseParallel = 'always'`).

Default: ''

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify Cost Function Evaluation Options

```
opt = sdo.EvaluateOptions;  
opt.Display = 'iter';  
opt.StopOnEvaluateError = 'on';
```

See Also

sdo.evaluate

Concepts

- Class Attributes
- Property Attributes

sdo.Experiment

Purpose Specify experiment I/O data, model parameters, and initial-state values

Description An *experiment* specifies input and output data for a Simulink model. You can also specify model parameters and initial-state values. Typically, you use experiments to estimate unknown model parameter values. You can also use the `createSimulator` method of an experiment to create a simulation object. Use the simulation object to simulate the model and compare measured and simulated data.

Construction `exp = sdo.Experiment(modelname)`

Constructs an `sdo.Experiment` object. It assigns the specified model name to the `ModelName` property and default values to the remaining properties.

Input Arguments

modelname

Simulink model name, specified as a string inside single quotes (' ').

The model must either be open or appear on the MATLAB path.

Properties

InitialStates

Model initial-state for the experiment, specified as a `param.State` object.

To specify multiple initial-states, use a vector of `param.State` objects.

To obtain model initial states, use `sdo.getStateFromModel`.

Use this property only for specifying initial-states that differ from the initial state values defined in the model.

- To estimate the value of an initial state, set the `Free` property of the initial state to `true`.

When you have multiple experiments for a given model, you can estimate model initial states on a per-experiment basis. To do so, specify the model initial states for each experiment. You can optionally specify an initial guess for the initial state values for any of the experiments using the `Value` property of the state parameters.

- To specify an initial state value as a known quantity, not to be estimated, set its `Free` property to `false`.

After specifying the initial states that you are estimating for an experiment, use `sdo.Experiment.getValuesToEstimate`. `sdo.Experiment.getValuesToEstimate` returns a vector of all the model parameters and initial states that you want to estimate. You use this vector as an input to `sdo.optimize` to specify the parameters that you want to estimate.

Default: []

InputData

Experiment input data.

Specify signals to apply to root-level input ports. For information on supported forms of input data, see “Forms of Input Data”.

Default: []

ModelName

Simulink model name associated with the experiment, specified as a string.

The model must appear on the MATLAB path.

Default: ''

OutputData

sdo.Experiment

Experiment output data, specified as a `Simulink.SimulationData.Signal` object.

To specify multiple output signals, use a vector of `Simulink.SimulationData.Signal` objects.

Default: []

Parameters

Model parameter value for the experiment, specified as a `param.Continuous` object.

To specify values for multiple parameters, use a vector of `param.Continuous` objects.

To obtain model parameters, use `sdo.getParameterFromModel`.

Use this property only for specifying parameters values that differ from the parameters values defined in the model.

- To estimate the value of a parameter, set the `Free` property of the parameter to `true`.

When you have multiple experiments for a given model, you can:

- Estimate a model parameter on a per-experiment basis. To do so, specify the model parameter for each experiment. You can optionally specify the initial guess for the parameter value for any of the experiments using the `Value` property.
- Estimate one value for a model parameter using all the experimental data. To do so, do not specify the model parameter for the experiments. Instead, call `sdo.optimize` with the model parameter directly.

For an example of estimating model parameters on a per-experiment basis and using data from multiple experiments, see “Estimate Model Parameters Per Experiment (Code)”.

- To specify a parameter value as a known quantity, not to be estimated, set its `Free` property to `false`.

After specifying the parameters that you are estimating for an experiment, use `sdo.Experiment.getValuesToEstimate`. `sdo.Experiment.getValuesToEstimate` returns a vector of all the model parameters and initial states that you want to estimate. You use this vector as an input to `sdo.optimize` to specify the parameters that you want to estimate.

Default: []

Name

Experiment name, specified as a string.

Default: ''

Description

Experiment description, specified as a string.

Default: ''

Methods

<code>createSimulator</code>	Create simulation object from experiment to compare measured and simulated data
<code>getValuesToEstimate</code>	Get model initial states and parameters for estimation from experiment
<code>setEstimatedValues</code>	Update experiments with estimated model initial states and parameter values

sdo.Experiment

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify Input and Output Data for Parameter Estimation

Load the measured experiment data.

```
load sdoBattery_ExperimentData
```

The variable `Charge_Data`, which contains the data measured during a battery charging experiment, is loaded into the MATLAB workspace. The first column contains time data. The second and third columns contain the current and voltage data.

Specify an experiment for a model.

```
modelName = 'sdoBattery';  
exp = sdo.Experiment(modelName);  
exp.Name = 'Charging';  
exp.Description = 'Battery charging data collected on March 15, 2013.';
```

Specify input data for the experiment.

```
exp.InputData = timeseries(Charge_Data(:,2),Charge_Data(:,1));
```

Specify output data for the experiment.

```
VoltageSig = Simulink.SimulationData.Signal;  
VoltageSig.Name = 'Voltage';  
VoltageSig.BlockPath = 'sdoBattery/SOC -> Voltage';  
VoltageSig.PortType = 'output';  
VoltageSig.PortIndex = 1;  
VoltageSig.Values = timeseries(Charge_Data(:,3),Charge_Data(:,1));
```

```
exp.OutputData = VoltageSig;
```

Alternatives

“Run Estimation”

See Also

[sdo.optimize](#) | [sdo.getStateFromModel](#) | [param.Continuous](#) | [param.State](#)

Related Examples

- [“Estimate Model Parameter Values \(Code\)”](#)
- [“Estimate Model Parameters and Initial States \(Code\)”](#)
- [“Estimate Model Parameters using Multiple Experiments \(Code\)”](#)
- [“Estimate Model Parameters Per Experiment \(Code\)”](#)
- [“Estimate Model Parameters with Parameter Constraints \(Code\)”](#)

Concepts

- [Class Attributes](#)
- [Property Attributes](#)

sdo.OptimizeOptions

Purpose	Optimization options
Syntax	<code>opt = sdo.OptimizeOptions</code> <code>opt = sdo.OptimizeOptions(Name,Value)</code>
Description	Specify options such as solver, solver options and whether to use parallel computing during optimization.
Construction	<code>opt = sdo.OptimizeOptions</code> creates an <code>sdo.OptimizeOptions</code> object and assigns default values to the properties. <code>opt = sdo.OptimizeOptions(Name,Value)</code> creates an <code>sdo.OptimizeOptions</code> object with additional options specified by one or more <code>Name,Value</code> pair arguments. <code>Name</code> is a property name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .

Input Arguments

Name-Value Pair Arguments

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

GradFcn

String that indicates whether the cost/constraint function you provide to `sdo.optimize` returns gradient information:

- 'on' — The cost/constraint function returns gradient information
- 'off' — The cost/constraint function does not return gradient information. The software uses central differences to compute the gradients.

Default: 'off'

Method

String specifying the optimization solver that `sdo.optimize` uses to solve the optimization problem:

- 'fmincon'
- 'fminsearch'
- 'lsqnonlin'
- 'patternsearch' (requires Global Optimization Toolbox software)

See the Optimization Toolbox and Global Optimization Toolbox documentation for more information on these solvers.

Default: 'fmincon'

MethodOptions

Structure with fields specifying optimization solver options. The structure fields are configured based on the `Method` property.

You can change solver options. For example
`opt.MethodOptions.TolX = 1.5e-3.`

For information on the optimization solver options, see:

- “Optimization Options” when `Method` is specified as 'fmincon', 'fminsearch', or 'lsqnonlin'
- `psoptimset` and “Pattern Search Options” when `Method` is specified as 'patternsearch'

Default: [1x1 struct]

OptimizedModel

sdo.OptimizeOptions

String displaying a Simulink model name to be optimized.

Default: ''

ParallelPathDependencies

Cell array of strings specifying path dependencies to use during parallel optimization. These path dependencies are temporarily added to the workers during parallel optimization.

Default: {}

Restarts

Nonnegative integer specifying the number of times the optimization solver restarts the optimization if convergence criteria are not satisfied. At each restart, the initial value of the tunable parameters are set to the final value of the previous optimization run.

Default: 0

StopIfFeasible

Terminate optimization once a feasible solution satisfying the constraints is found:

- 'on' — Terminate as soon a feasible solution is found
- 'off' — Continue to search for solutions that are typically located further inside the constraint region

The software ignores this option when you track a reference signal or your problem has a cost.

Default: 'on'

UseParallel

Parallel computing option for `fmincon`, `lsqnonlin` and `patternsearch` optimization solvers:

- 'never' — Do not use parallel computing during optimization
- 'always' — Use parallel computing during optimization

Parallel Computing Toolbox software must be installed to enable parallel computing for the optimization methods.

When set to 'always', the methods compute the following in parallel:

- `fmincon` — Finite difference gradients
- `lsqnonlin` — Finite difference gradients
- `patternsearch` — Population evaluation

Default: 'never'

Properties

GradFcn

String that indicates whether the cost/constraint function you provide to `sdo.optimize` returns gradient information:

- 'on' — The cost/constraint function returns gradient information
- 'off' — The cost/constraint function does not return gradient information. The software uses central differences to compute the gradients.

Default: 'off'

Method

String specifying the optimization solver that `sdo.optimize` uses to solve the optimization problem:

sdo.OptimizeOptions

- 'fmincon'
- 'fminsearch'
- 'lsqnonlin'
- 'patternsearch' (requires Global Optimization Toolbox software)

See the Optimization Toolbox and Global Optimization Toolbox documentation for more information on these solvers.

Default: 'fmincon'

MethodOptions

Structure with fields specifying optimization solver options. The structure fields are configured based on the Method property.

You can change solver options. For example
`opt.MethodOptions.TolX = 1.5e-3.`

For information on the optimization solver options, see:

- “Optimization Options” when Method is specified as 'fmincon', 'fminsearch', or 'lsqnonlin'
- `psoptimset` and “Pattern Search Options” when Method is specified as 'patternsearch'

Default: [1x1 struct]

OptimizedModel

String displaying a Simulink model name to be optimized.

Default: ''

ParallelPathDependencies

Cell array of strings specifying path dependencies to use during parallel optimization. These path dependencies are temporarily added to the workers during parallel optimization.

Default: {}

Restarts

Nonnegative integer specifying the number of times the optimization solver restarts the optimization if convergence criteria are not satisfied. At each restart, the initial value of the tunable parameters are set to the final value of the previous optimization run.

Default: 0

StopIfFeasible

Terminate optimization once a feasible solution satisfying the constraints is found:

- 'on' — Terminate as soon a feasible solution is found
- 'off' — Continue to search for solutions that are typically located further inside the constraint region

The software ignores this option when you track a reference signal or your problem has a cost.

Default: 'on'

UseParallel

Parallel computing option for `fmincon`, `lsqnonlin` and `patternsearch` optimization solvers:

- 'never' — Do not use parallel computing during optimization
- 'always' — Use parallel computing during optimization

sdo.OptimizeOptions

Parallel Computing Toolbox software must be installed to enable parallel computing for the optimization methods.

When set to 'always', the methods compute the following in parallel:

- `fmincon` — Finite difference gradients
- `lsqnonlin` — Finite difference gradients
- `patternsearch` — Population evaluation

Default: 'never'

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify an optimization solver.

```
opts = sdo.OptimizeOptions;  
opts.Method = 'fminsearch';
```

See Also

`sdo.optimize` | `sdo.getModelDependencies`

How To

- “Optimization Options”
- “Speedup Using Parallel Computing”

Purpose

Specify probability distributions for model parameters

Description

Specify the probability distributions for model parameters, which define the *parameter space*. You use the `sdo.ParameterSpace` object as an input to the `sdo.sample` command and generate samples of the model parameters. The software generates these samples as per the distributions specified for each parameter. You evaluate the cost function for each of these samples using the `sdo.evaluate` command and analyze how the model parameters influence the cost function.

Construction

`ps = sdo.ParameterSpace(p)` creates an `sdo.ParameterSpace` object for the specified model parameters. The software assigns the parameter names to the `ParameterNames` property and default values to the remaining properties, including `ParameterDistributions`. The software specifies the uniform distribution for each parameter in `p` and sets the values of the two parameters of the uniform distribution as follows:

- **Lower** — Set to `p.Minimum`. If `p.Minimum` is equal to `-Inf`, then the software sets `Lower` to `0.9*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Lower` to `-1`.
- **Upper** — Set to `p.Maximum`. If `p.Maximum` is equal to `Inf`, then the software sets `Upper` to `1.1*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Upper` to 1.

`ps = sdo.ParameterSpace(p,pdist)` specifies the distribution of each parameter.

Input Arguments

p

Model parameters and states, specified as a vector of `param.Continuous` objects.

For example,

```
sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'}).
```

sdo.ParameterSpace

pdist

Probability distribution of model parameters, specified as a vector of univariate probability distribution objects.

- If `pdist` is the same size as `p`, the software specifies each entry of `pdist` as the probability distribution of the corresponding parameter in `p`.
- If `pdist` contains only one object, the software specifies this object as the probability distribution for all the parameters in `p`.

Use the `makedist` command to create a univariate probability distribution object. For example, `makedist('Normal','mu',100,'sigma',10)`.

To check if `pdist` is a univariate distribution object, run `isa(pdist,'prob.UnivariateDistribution')`.

Properties

ParameterNames

Model parameter names, specified as cell arrays of strings.

This property is read only.

Default: ''

ParameterDistributions

Model parameter distributions, specified as a vector of `prob.UnivariateDistribution` objects.

By default, the software specifies a uniform distribution for the model parameters specified by `p`. For each parameter, the software sets the values of the two parameters of the uniform distribution:

- `Lower` — Set to `p.Minimum`. If `p.Minimum` is equal to `-Inf`, then the software sets `Lower` to `0.9*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Lower` to `-1`.

- **Upper** — Set to `p.Maximum`. If `p.Maximum` is equal to `Inf`, then the software sets `Upper` to `1.1*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Upper` to 1.

Use the `pdist` input argument when constructing `ps` to set the value of this property. Alternatively, use the `sdo.ParameterSpace.setDistribution` method after you have constructed `ps`.

Default: []

RankCorrelation

Correlation between parameters, specified as a matrix.

When you call `sdo.sample`, the software generates samples that are correlated as specified by this matrix (where the correlation refers to ranked correlation). You can specify the sampling method using the `Method` property of an `sdo.SampleOptions`.

- If you specify `Method` as `'random'` or `'lhs'`, the software uses the Iman-Conover algorithm to impose the correlation specified by `RankCorrelation`.
- If you specify `Method` as `'copula'`, the software uses a copula to impose the correlation specified by `RankCorrelation`. Use the `MethodOptions` property of the `sdo.SampleOptions` object to specify the copula family.

Specify [] when the parameters are uncorrelated.

Default: []

Options

Sampling method options, specified as an `sdo.SampleOptions` object.

Default: `sdo.SampleOptions`

sdo.ParameterSpace

Notes

Text notes associated with `ps`, specified as a string or cell array of strings.

Default: ''

Methods

<code>addParameter</code>	Add parameter to <code>sdo.ParameterSpace</code> object
<code>removeParameter</code>	Remove parameter from <code>sdo.ParameterSpace</code> object
<code>setDistribution</code>	Set distribution of parameter in <code>sdo.ParameterSpace</code> object

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify Parameter Distributions for Sampling

Obtain the model parameters of interest.

```
load_system('sdoHydraulicCylinder');  
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

Construct an `sdo.ParameterSpace` object for `Ac` and `K`.

```
ps = sdo.ParameterSpace(p);
```

You can use `ps` as an input to `sdo.sample` and generate samples. By default, the software specifies a uniform distribution for both parameters.

Suppose you want to specify the normal distribution for `Ac` and the uniform distribution for `K`, with `K` in the [30000 70000] range.

```
pdistAc = makedist('Normal', 'mu',p(1).Value,'sigma',2);
```

```
pdistK = makedist('Uniform','lower',30000,'upper',70000);  
ps1 = sdo.ParameterSpace(p,[pdistAc;pdistK]);
```

See Also

[sdo.ParameterSpace.addParameter](#) | [sdo.sample](#) |
[sdo.getParameterFromModel](#) | [makedist](#)

Concepts

- [Class Attributes](#)
- [Property Attributes](#)

sdo.requirements.BodeMagnitude

Purpose Bode magnitude bound

Syntax
`bode_req = sdo.requirements.BodeMagnitude`
`bode_req = sdo.requirements.BodeMagnitude(Name,Value)`

Description Specify frequency-dependent piecewise-linear upper and lower magnitude bounds on a linear system. You can then optimize your model to meet the requirements using `sdo.optimize`.

You can specify upper or lower bounds, include multiple linear edges, and extend them to + or -infinity..

You must have Simulink Control Design software to specify bode magnitude requirements.

Construction `bode_req = sdo.requirements.BodeMagnitude` creates an `sdo.requirements.BodeMagnitude` object and assigns default values to its properties.

`bode_req = sdo.requirements.BodeMagnitude(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

BoundFrequencies

Frequency values for the gain bound.

Specify the start and end frequencies for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles, where each row specifies the start and end frequencies of an edge in the piecewise-linear bound. The start and end frequencies must define a positive length. The number of rows must match the number of rows of the `BoundMagnitudes` property.

Use `set` to set this and the `BoundMagnitudes` properties simultaneously.

Use the `FrequencyUnits` property to specify the frequency units.

Default: [1 10]

BoundMagnitudes

Magnitude values for the gain bound.

Specify the start and end gain values for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles where each row specifies the start and end gains of an edge in the piecewise-linear bound. The number of rows must match the number of rows of the `BoundFrequencies` property.

Use `set` to set this and the `BoundFrequencies` properties simultaneously.

Use the `MagnitudeUnits` property to specify the magnitude units.

Default: [0 0]

Description

Requirement description. Must be a string.

Default: ''

FrequencyScale

sdo.requirements.BodeMagnitude

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points. Must be one of the following strings:

- 'linear'
- 'log'

For example, if bound edges are at frequencies f_1 and f_2 , and the bound is to be evaluated at f_3 , the edges are interpolated as a straight lines. The x-axis is either linear or logarithmic.

Default: 'log'

FrequencyUnits

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'
- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'

- 'rad/week'
- 'rad/month'
- 'rad/year'
- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'
- 'cycles/year'

Default: 'rad/s'

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive frequency direction.

sdo.requirements.BodeMagnitude

Specify whether the first and last edge of the bound extends to `-inf` and `+inf` respectively. Use to bound signals that extend beyond the frequency values specified by the `BoundFrequencies` property.

Must be a 1x2 logical array of `true` or `false`. If `true`, the first or last edge of the piecewise linear bound is extended in the negative or positive direction.

Default: [0 0]

Type

Magnitude bound type. Must be:

- '`<=`' — Upper bound
- '`>=`' — Lower bound

Use to specify whether the piecewise-linear bound is an upper or lower bound. Use for upper bound and for lower bound.

Properties

BoundFrequencies

Frequency values for the gain bound.

Specify the start and end frequencies for all the edges in the piecewise-linear bound. The property must be a `nx2` array of finite doubles, where each row specifies the start and end frequencies of an edge in the piecewise-linear bound. The start and end frequencies must define a positive length. The number of rows must match the number of rows of the `BoundMagnitudes` property.

Use `set` to set this and the `BoundMagnitudes` properties simultaneously.

Use the `FrequencyUnits` property to specify the frequency units.

Default: [1 10]

BoundMagnitudes

Magnitude values for the gain bound.

Specify the start and end gain values for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles where each row specifies the start and end gains of an edge in the piecewise-linear bound. The number of rows must match the number of rows of the `BoundFrequencies` property.

Use `set` to set this and the `BoundFrequencies` properties simultaneously.

Use the `MagnitudeUnits` property to specify the magnitude units.

Default: [0 0]

Description

Requirement description. Must be a string.

Default: ''

FrequencyScale

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points. Must be one of the following strings:

- 'linear'
- 'log'

For example, if bound edges are at frequencies f_1 and f_2 , and the bound is to be evaluated at f_3 , the edges are interpolated as a straight lines. The x-axis is either linear or logarithmic.

Default: 'log'

FrequencyUnits

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'
- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'
- 'rad/week'
- 'rad/month'
- 'rad/year'
- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'

- 'cycles/year'

Default: 'rad/s'

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive frequency direction.

Specify whether the first and last edge of the bound extends to `-inf` and `+inf` respectively. Use to bound signals that extend beyond the frequency values specified by the `BoundFrequencies` property.

Must be a 1x2 logical array of `true` or `false`. If `true`, the first or last edge of the piecewise linear bound is extended in the negative or positive direction.

Default: [0 0]

Type

Magnitude bound type. Must be:

- '<=' — Upper bound

sdo.requirements.BodeMagnitude

- '>=' — Lower bound

Use to specify whether the piecewise-linear bound is an upper or lower bound. Use for upper bound and for lower bound.

Methods

evalRequirement	Evaluate Bode magnitude bound for linear system
-----------------	---

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a Bode magnitude requirements object and specify bound frequencies and magnitudes.

```
r = sdo.requirements.BodeMagnitude;  
set(r,'BoundFrequencies', [0.1 10; 10 100],...  
'BoundMagnitudes',[1 1; 0.1 0.1])
```

Alternatively, you can specify the frequency and magnitude during construction.

```
r = sdo.requirements.BodeMagnitude(...  
'BoundFrequencies', [1 10; 10 100], ...  
'BoundMagnitudes', [1 1; 1 0]);
```

Alternatives

Use `getbounds` to get the bounds specified in a Check Bode Characteristics block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

Purpose	Closed loop peak gain bound
Description	<p>Specify lower or equality bounds on the closed loop peak gain of a linear system. The closed loop can be formed using negative, positive or no feedback. You can then optimize the model response to meet these bounds using <code>sdo.optimize</code>.</p> <p>You must have Simulink Control Design software to specify closed-loop peak gain bounds.</p>
Construction	<p><code>pkgain_req = sdo.requirements.ClosedLoopPeakGain</code> creates a <code>sdo.requirements.ClosedLoopPeakGain</code> object and assigns default values to its properties.</p> <pre>pkgain_req = sdo.requirements.ClosedLoopPeakGain(Name,Value)</pre> uses additional options specified by one or more <code>Name,Value</code> pair arguments. <code>Name</code> is a property name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (<code>' '</code>). You can specify several name-value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .
	Input Arguments
	Name-Value Pair Arguments
	Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (<code>' '</code>). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .
	Description
	Requirement description. Must be a string.
	Default: <code>''</code>
	FeedbackSign

sdo.requirements.ClosedLoopPeakGain

Feedback loop sign to determine the peak gain of the linear system.

Must be -1 or 1 . Use -1 if the loop has negative feedback and 1 if the loop has positive feedback.

Default: 1

MagnitudeUnits

Magnitude units of the requirement.

Must be 'db' (decibels) or 'abs' (absolute units).

Default: 'abs'

Name

Requirement name. Must be a string.

Default: ''

PeakGain

Peak gain bound.

Default: 2

Type

Peak gain requirement type. Must be one of the following strings:

- '<=' — Upper bound
- '==' — Equality bound
- 'min' — Minimization objective

Default: '<='

Properties

Description

Requirement description. Must be a string.

Default: ''

FeedbackSign

Feedback loop sign to determine the peak gain of the linear system.

Must be -1 or 1. Use -1 if the loop has negative feedback and 1 if the loop has positive feedback.

Default: 1

MagnitudeUnits

Magnitude units of the requirement.

Must be 'db' (decibels) or 'abs' (absolute units).

Default: 'abs'

Name

Requirement name. Must be a string.

Default: ''

PeakGain

Peak gain bound.

Default: 2

Type

Peak gain requirement type. Must be one of the following strings:

- '<=' — Upper bound

sdo.requirements.ClosedLoopPeakGain

- '==' — Equality bound
- 'min' — Minimization objective

Default: '<='

Methods

`evalRequirement` Evaluate peak gain bound for linear system

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a closed loop peak gain object and specify peak gain requirement.

```
r = sdo.requirements.ClosedLoopPeakGain;  
r.PeakGain = 2;
```

Alternatively, you can specify the peak gain during construction:

```
r = sdo.requirements.ClosedLoopPeakGain('PeakGain',2);
```

Alternatives

Use `getbounds` to get the bounds specified in Check Nichols Characteristics block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

Purpose	Gain and phase margin bounds
Description	<p>Specify lower or equality bounds on the gain and phase margin of a linear system. You can then optimize the model response to meet the bounds using <code>sdo.optimize</code>.</p> <p>You must have Simulink Control Design software to specify gain and phase margin requirements.</p>
Construction	<p><code>gainphase_req = sdo.requirements.GainPhaseMargin</code> creates a <code>sdo.requirements.GainPhaseMargin</code> object and assigns default values to its properties.</p> <pre>gainphase_req = sdo.requirements.GainPhaseMargin(Name,Value)</pre> uses additional options specified by one or more <code>Name,Value</code> pair arguments. <code>Name</code> is a property name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (<code>' '</code>). You can specify several name-value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .
	Input Arguments
	Name-Value Pair Arguments
	Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes (<code>' '</code>). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .
	Description
	Requirement description. Must be a string.
	Default: <code>''</code>
	FeedbackSign
	Feedback loop sign to determine the gain and phase margins of the linear system.

sdo.requirements.GainPhaseMargin

Must be -1 or 1 . Use -1 if the loop has negative feedback and 1 if the loop has positive feedback.

Default: 1

GainMargin

Gain margin bound. Use `MagnitudeUnits` to specify the gain units. Set to `[]` to specify a bound on the phase margin only.

Default: `10`

MagnitudeUnits

Magnitude units of the requirement. Must be:

- `'db'` (decibels)
- `'abs'` (absolute units)

Default: `'db'`

Name

Requirement name. Must be a string.

Default: `''`

PhaseMargin

Phase margin bound. Must be in degrees and a positive finite scalar. Set to `[]` to specify a bound on the gain margin only.

Default: `60`

PhaseUnits

Phase units of the requirement. Must be one of the following strings:

- 'deg' (degrees)
- 'rad' (radians)

Default: 'deg'

Type

Gain and phase margin requirement type. Must be one of the following strings:

- '>=' — Lower bound
- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Properties

Description

Requirement description. Must be a string.

Default: ''

FeedbackSign

Feedback loop sign to determine the gain and phase margins of the linear system.

Must be -1 or 1. Use -1 if the loop has negative feedback and 1 if the loop has positive feedback.

Default: 1

GainMargin

Gain margin bound. Use `MagnitudeUnits` to specify the gain units. Set to `[]` to specify a bound on the phase margin only.

Default: 10

sdo.requirements.GainPhaseMargin

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

PhaseMargin

Phase margin bound. Must be in degrees and a positive finite scalar. Set to [] to specify a bound on the gain margin only.

Default: 60

PhaseUnits

Phase units of the requirement. Must be one of the following strings:

- 'deg' (degrees)
- 'rad' (radians)

Default: 'deg'

Type

Gain and phase margin requirement type. Must be one of the following strings:

- '>=' — Lower bound

- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Methods

`evalRequirement` Evaluate gain and phase margin bounds for linear system

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a gain and phase margin object and specify gain and phase margin requirement.

```
r = sdo.requirements.GainPhaseMargin;  
r.GainMargin = 5;  
r.PhaseMargin = 55;
```

Alternatively, you can specify the gain and phase margins during construction.

```
r = sdo.requirements.GainPhaseMargin(...  
    'GainMargin',5, ...  
    'PhaseMargin', 55);
```

Alternatives

Use `getbounds` to get the bounds specified in a Check Gain and Phase Margins and Check Nichols Characteristics block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

sdo.requirements.OpenLoopGainPhase

Purpose Nichols response bound

Description Specify piecewise-linear bounds on the Nichols (gain-phase) response of a linear system. You can then optimize the model response to meet these bounds using `sdo.optimize`.

You can specify an upper or lower bound, include multiple linear edges, and extend the bounds to `+` or `inf`.

You must have Simulink Control Design software to specify open-loop gain and phase requirements.

Construction `olgainphase_req = sdo.requirements.OpenLoopGainPhase` creates a `sdo.requirements.OpenLoopGainPhase` object and assigns default values to its properties.

`gainphase_req = sdo.requirements.OpenLoopGainPhase(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

BoundGains

Gain values for a piecewise linear bound.

Specify the start and end values in decibels for all the edges in the piecewise-linear bound. The property must be a `nx2` array of finite doubles, where each row specifies the start and end gain values

of an edge. The number of rows must match the number of rows of the **BoundPhases** property.

Use **set** to set this and the **BoundPhases** properties simultaneously.

Default: [-10 -10]

BoundPhases

Phase values for a piecewise-linear bound.

Specify the start and end values in degrees for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles, where each row specifies the start and end phase values of an edge. The number of rows must match the number of rows of the **BoundGains** property.

Use **set** to set this and the **BoundGains** properties simultaneously.

Default: [-180 -90]

Description

Requirement description. Must be a string.

Default: ''

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

sdo.requirements.OpenLoopGainPhase

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive time direction.

Use to bound signals that extend beyond the coordinates specified by the **BoundPhases** and **BoundGains** properties.

Must be a 1x2 logical array. If **true**, the first or last edge of the bound is extended to infinity.

Default: [0 0]

PhaseUnits

Phase units of the requirement. Must be one of the following strings:

- 'deg' (degrees)
- 'rad' (radians)

Default: 'deg'

Type

Gain and phase requirement type. Must be one of the following strings:

- '>=' — Lower bound
- '<=' — Upper bound

Default: '>='

Properties

BoundGains

Gain values for a piecewise linear bound.

Specify the start and end values in decibels for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles, where each row specifies the start and end gain values of an edge. The number of rows must match the number of rows of the **BoundPhases** property.

Use **set** to set this and the **BoundPhases** properties simultaneously.

Default: [-10 -10]

BoundPhases

Phase values for a piecewise-linear bound.

Specify the start and end values in degrees for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles, where each row specifies the start and end phase values of an edge. The number of rows must match the number of rows of the **BoundGains** property.

Use **set** to set this and the **BoundGains** properties simultaneously.

Default: [-180 -90]

Description

Requirement description. Must be a string.

Default: ''

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

sdo.requirements.OpenLoopGainPhase

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive time direction.

Use to bound signals that extend beyond the coordinates specified by the **BoundPhases** and **BoundGains** properties.

Must be a 1x2 logical array. If **true**, the first or last edge of the bound is extended to infinity.

Default: [0 0]

PhaseUnits

Phase units of the requirement. Must be one of the following strings:

- 'deg' (degrees)
- 'rad' (radians)

Default: 'deg'

Type

Gain and phase requirement type. Must be one of the following strings:

- '>=' — Lower bound
- '<=' — Upper bound

Default: '>='

Methods

`evalRequirement` Evaluate gain and phase bounds on Nichols response of linear system

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct an open-loop gain and phase object, and specify gain and phase requirements.

```
r = sdo.requirements.OpenLoopGainPhase;  
set(r,'BoundPhases',[-120 -120; -120 -150; -150 -180],...  
    'BoundGains',[20 0; 0 -20; -20 -20]);
```

Alternatively, you can specify the gain and phase requirements during construction:

```
r = sdo.requirements.OpenLoopGainPhase('BoundPhases',...  
    [-120 -120; -120 -150; -150 -180],'BoundGains',...  
    [20 0; 0 -20; -20 -20]);
```

Alternatives

Use `getbounds` to get the bounds specified in a Check Nichols Characteristics block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

sdo.requirements.PZDampingRatio

Purpose Damping ratio bound

Description Specify bounds on the damping ratio of the poles of a linear system. You can then optimize the model response to meet these bounds using `sdo.optimize`. You can also use this object to specify overshoot bound. You must have Simulink Control Design software to specify damping ratio requirements.

Construction `damp_req = sdo.requirements.PZDampingRatio` creates a `sdo.requirements.PZDampingRatio` object and assigns default values to its properties.

`gainphase_req = sdo.requirements.PZDampingRatio(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

DampingRatio

Damping ratio bound. Must be a finite scalar between 0 and 1.

Default: 0.7071

Description

Requirement description. Must be a string.

Default: ''

Name

Requirement name. Must be a string.

Default: ''

Type

Damping ratio bound type. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound
- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Properties**DampingRatio**

Damping ratio bound. Must be a finite scalar between 0 and 1.

Default: 0.7071

Description

Requirement description. Must be a string.

Default: ''

Name

Requirement name. Must be a string.

Default: ''

Type

Damping ratio bound type. Must be one of the following strings:

sdo.requirements.PZDampingRatio

- '<=' — Upper bound
- '>=' — Lower bound
- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Methods

evalRequirement	Evaluate damping ratio bound on linear system
getOvershoot	Convert damping ratio to equivalent overshoot value
setOvershoot	Set overshoot to an equivalent damping ratio

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a damping ratio object and specify the damping ratio.

```
r = sdo.requirements.PZDampingRatio;  
r.DampingRatio = 0.1;
```

Alternatively, you can specify the damping ratio during construction.

```
r = sdo.requirements.PZDampingRatio('DampingRatio',0.1);
```

Alternatives

Use getbounds to get the bounds specified in a Check Pole-Zero Characteristics block.

See Also

copy | get | set

How To

- Class Attributes
- Property Attributes

sdo.requirements.PZNaturalFrequency

Purpose Natural frequency bound

Description Specify bounds on the natural frequency of the poles of a linear system. You can then optimize the model response to meet these bounds using `sdo.optimize`.

You must have Simulink Control Design software to specify natural frequency requirements.

Construction `pznatfreq_req = sdo.requirements.PZNaturalFrequency` creates a `sdo.requirements.PZNaturalFrequency` object and assigns default values to its properties.

`pznatfreq_req = sdo.requirements.pznatfreq_req(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

Description

Requirement description. Must be a string.

Default: `' '`

FrequencyUnits

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'
- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'
- 'rad/week'
- 'rad/month'
- 'rad/year'
- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'
- 'cycles/year'

Default: 'rad/s'

sdo.requirements.PZNaturalFrequency

Name

Requirement name. Must be a string.

Default: ''

NaturalFrequency

Natural frequency bound. Must be in radians/second and a positive finite scalar.

Default: 2

Type

Natural frequency bound type. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound
- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Properties

Description

Requirement description. Must be a string.

Default: ''

FrequencyUnits

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'

- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'
- 'rad/week'
- 'rad/month'
- 'rad/year'
- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'
- 'cycles/year'

Default: 'rad/s'

Name

sdo.requirements.PZNaturalFrequency

Requirement name. Must be a string.

Default: ''

NaturalFrequency

Natural frequency bound. Must be in radians/second and a positive finite scalar.

Default: 2

Type

Natural frequency bound type. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound
- '==' — Equality bound
- 'max' — Maximization objective

Default: '>='

Methods

evalRequirement	Evaluate natural frequency bound on linear system
-----------------	---

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a natural frequency object and specify the natural frequency.

```
r = sdo.requirements.PZNaturalFrequency;  
r.NaturalFrequency = 1;
```

Alternatively, you can specify the natural frequency during construction.


```
r = sdo.requirements.PZNaturalFrequency(...  
    'NaturalFrequency',1);
```

Alternatives Use `getbounds` to get the bounds specified in a Check Pole-Zero Characteristics block.

See Also `copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

sdo.requirements.PZSettlingTime

Purpose Settling time bound

Description Specify bounds on the real component of the poles of a linear system. The real component of poles are used to approximate the settling time. You can then optimize the model response to meet these bounds using `sdo.optimize`.

You must have Simulink Control Design software to specify settling time requirements.

Construction `settime_req = sdo.requirements.PZSettlingTime` creates a `sdo.requirements.PZSettlingTime` object and assigns default values to its properties.

`settime_req = sdo.requirements.PZSettlingTime(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

Description

Requirement description. Must be a string.

Default: ' '

Name

Requirement name. Must be a string.

Default: ''

SettlingTime

Settling time bound. Must be in seconds and a positive finite scalar.

Default: 2

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Settling time bound type. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound

sdo.requirements.PZSettlingTime

- '==' — Equality bound
- 'min' — Minimization objective

Default: '<='

Properties

Description

Requirement description. Must be a string.

Default: ''

Name

Requirement name. Must be a string.

Default: ''

SettlingTime

Settling time bound. Must be in seconds and a positive finite scalar.

Default: 2

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'

- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Settling time bound type. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound
- '==' — Equality bound
- 'min' — Minimization objective

Default: '<='

Methods

evalRequirement	Evaluate settling time bound on linear system
-----------------	---

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a settling time object and specify the settling time requirement.

```
r = sdo.requirements.PZSettlingTime;  
r.SettlingTime = 2.5;
```

Alternatively, you can specify the setting time during construction.

```
r = sdo.requirements.PZSettlingTime('SettlingTime',2.5);
```

sdo.requirements.PZSettlingTime

Alternatives Use getbounds to get the bounds specified in a Check Pole-Zero Characteristics block.

See Also copy | get | set

How To

- Class Attributes
- Property Attributes

Purpose Piecewise-linear amplitude bound

Description Specify piecewise-linear upper or lower amplitude bounds on a time-domain signal. You can then optimize the model response to meet these bounds using `sdo.optimize`.

You can include multiple linear edges, and extend to $+$ or $-\infty$.

Construction `sig_req = sdo.requirements.SignalBound` creates an `sdo.requirements.SignalBound` object and assigns default values to its properties.

`sig_req = sdo.requirements.SignalBound(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

BoundMagnitudes

Magnitude values for the piecewise-linear bound.

Specify the start and end magnitude values for all edges in the bound. The property must be a `nx2` array of finite doubles, where each row specifies the start and end magnitude values of an edge. The number of rows must match the number of rows of the `BoundTimes` property.

Use `set` to set this and `BoundTimes` properties simultaneously.

sdo.requirements.SignalBound

Default: [1 1]

BoundTimes

Time values of the piecewise-linear bound.

Specify the start and end times for all the edges in the piecewise-linear bound. The property must be a nx2 array of finite doubles where each row specifies the start and end times of an edge. The start and end times must define a positive length. The number of rows must match the number of rows of the **BoundMagnitudes** property.

Use **set** to set this and **BoundMagnitudes** properties simultaneously.

Default: [0 10]

Description

Requirement description. Must be a string.

Default: ''

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive time direction.

Specify whether the first and last edge of the bound extends to **inf** and **+inf** respectively. Use to bound signals that extend beyond the time values specified by the **BoundTimes** property.

Must be a 1x2 logical array. If **true**, the first or last edge of the bound is extended in a negative or positive direction, respectively.

Default: [0 0]

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Bound type.

Specify whether the piecewise-linear requirement is an upper or lower bound. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound

Default: '<='

sdo.requirements.SignalBound

Properties

BoundMagnitudes

Magnitude values for the piecewise-linear bound.

Specify the start and end magnitude values for all edges in the bound. The property must be a $n \times 2$ array of finite doubles, where each row specifies the start and end magnitude values of an edge. The number of rows must match the number of rows of the `BoundTimes` property.

Use `set` to set this and `BoundTimes` properties simultaneously.

Default: [1 1]

BoundTimes

Time values of the piecewise-linear bound.

Specify the start and end times for all the edges in the piecewise-linear bound. The property must be a $n \times 2$ array of finite doubles where each row specifies the start and end times of an edge. The start and end times must define a positive length. The number of rows must match the number of rows of the `BoundMagnitudes` property.

Use `set` to set this and `BoundMagnitudes` properties simultaneously.

Default: [0 10]

Description

Requirement description. Must be a string.

Default: ''

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive time direction.

Specify whether the first and last edge of the bound extends to `inf` and `+inf` respectively. Use to bound signals that extend beyond the time values specified by the `BoundTimes` property.

Must be a 1x2 logical array. If `true`, the first or last edge of the bound is extended in a negative or positive direction, respectively.

Default: [0 0]

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Bound type.

sdo.requirements.SignalBound

Specify whether the piecewise-linear requirement is an upper or lower bound. Must be one of the following strings:

- '<=' — Upper bound
- '>=' — Lower bound

Default: '<='

Methods

evalRequirement Evaluate piecewise-linear bound

Examples

Construct a signal bound object and specify piecewise-linear bounds.

```
r = sdo.requirements.SignalBound;  
set(r, 'BoundTimes', [0 10; 10 20], ...  
     'BoundMagnitudes', [1.1 1.1; 1.01 1.01])
```

Alternatively, you can specify the bounds during construction:

```
r = sdo.requirements.SignalBound(...  
     'BoundTimes', [0 10; 10 20], ...  
     'BoundMagnitudes', [1.1 1.1; 1.01 1.01]);
```

Alternatives

Use `getbounds` to get the bounds specified in a Check Custom Bounds block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

Purpose Reference signal to track

Description Specify a tracking requirement on a time-domain signal. You can then optimize the model response to track the reference using `sdo.optimize`. You can specify an equality, upper or lower bound requirement.

Construction `track_req = sdo.requirements.SignalTracking` creates an `sdo.requirements.SignalTracking` object and assigns default values to its properties.

`track_req = sdo.requirements.SignalTracking(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

AbsTol

Absolute tolerance used to determine bounds as the signal approaches the reference signal. The bounds on the reference signal are given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

where y_r is the value of the reference at a certain time, y_u and y_l are the upper and lower tolerance bounds corresponding to that time point.

sdo.requirements.SignalTracking

Default: 0

Description

Requirement description. Must be a string.

Default: ''

InterpolationTimes

Time points to use when comparing reference and testpoint signals. Linear interpolation is used to compare the signals at the same timepoints.

Must be one of the following strings:

- 'Reference only' — Compare the signals at the time points of the reference signal only
- 'Testpoint only' — Compare the signals at the time points of the testpoint signal only
- 'Reference and Testpoint' — Compare the signals at the time points of both the reference and testpoint signals

Default: 'Reference only'

Method

Algorithm for evaluating the requirement when the **Type** property is '=='.

When the requirement is evaluated using `evalRequirement`, the software computes the error between the reference and testpoint signals. This property specifies how the error signal $e(t) = y_s(t) - y_r(t)$ should be processed.

Must be one of the following strings:

- 'SSE'

- 'SAE'
- 'Residuals'

Default: 'SSE'

Name

Requirement name. Must be a string.

Default: ''

Normalize

Enable or disable normalization when evaluating the requirement. The maximum absolute value of the reference signal is used for normalization. Must be 'on' or 'off'.

Default: 'on'

ReferenceSignal

Reference signal to track. Must be a MATLAB `timeseries` object with real finite data points.

Default: [1x1 timeseries]

RelTol

Relative tolerance used to determine bounds as the signal approaches the reference signal. The bounds on the reference signal are given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

Default: 0

RobustCost

sdo.requirements.SignalTracking

Enable or disable robust treatment of outliers when evaluating the requirement. The software uses a Huber loss function to handle the outliers in the cost function and improves the fit quality. This option reduces the influence of outliers on the estimation without you manually modifying your data.

Must be one of the following:

- 'on' — When you call the `evalRequirement` method, the software uses a Huber loss function to evaluate the cost for the tracking error outliers. The tracking error is calculated as $e(t) = y_{ref}(t) - y_{test}(t)$. The software uses the error statistics to identify the outliers.

The exact cost function used, $F(x)$, depends on the requirement evaluation Method.

Method Name	Cost Function for Nonoutliers	Cost Function for Outliers
'SSE'	$F(x) = \sum_{t \in NOL} e(t)^2$ <p><i>NOL</i> is the set of nonoutlier samples.</p>	$F(x) = \sum_{t \in OL} w \times e(t) $ <p><i>w</i> is a linear weight. <i>OL</i> is the set of outlier samples.</p>
'SAE'	$F(x) = \sum_{t \in NOL} e(t) $ <p><i>NOL</i> is the set of nonoutlier samples.</p>	$F(x) = \sum_{t \in OL} w$ <p><i>w</i> is a constant value. <i>OL</i> is the set of outlier samples.</p>
'Residuals'	The software does not remove the outliers.	

Method Name	Cost Function for Nonoutliers	Cost Function for Outliers
	$F(x) = \begin{bmatrix} e(0) \\ \vdots \\ e(N) \end{bmatrix}$ <p>N is the number of samples.</p>	

- 'off'

Default: 'off'

Type

Tracking requirement type. Must be one of the following strings:

- '=' — Tracking objective.
- '<=' — Upper bound
- '>=' — Lower bound

Default: '='

Weights

Weights to use when evaluating the tracking error between the reference and testpoint signals. Use weights to increase or decrease the significance of different time points.

Must be real finite positive vector with the same number of elements as the `Time` property of the MATLAB `timeseries` object in the `ReferenceSignal` property.

Properties

AbsTol

Absolute tolerance used to determine bounds as the signal approaches the reference signal. The bounds on the reference signal are given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

where y_r is the value of the reference at a certain time, y_u and y_l are the upper and lower tolerance bounds corresponding to that time point.

Default: 0

Description

Requirement description. Must be a string.

Default: ''

InterpolationTimes

Time points to use when comparing reference and testpoint signals. Linear interpolation is used to compare the signals at the same timepoints.

Must be one of the following strings:

- 'Reference only' — Compare the signals at the time points of the reference signal only
- 'Testpoint only' — Compare the signals at the time points of the testpoint signal only
- 'Reference and Testpoint' — Compare the signals at the time points of both the reference and testpoint signals

Default: 'Reference only'

Method

Algorithm for evaluating the requirement when the Type property is '=='.

When the requirement is evaluated using `evalRequirement`, the software computes the error between the reference and testpoint signals. This property specifies how the error signal $e(t) = y_s(t) - y_r(t)$ should be processed.

Must be one of the following strings:

- 'SSE'
- 'SAE'
- 'Residuals'

Default: 'SSE'

Name

Requirement name. Must be a string.

Default: ''

Normalize

Enable or disable normalization when evaluating the requirement. The maximum absolute value of the reference signal is used for normalization. Must be 'on' or 'off'.

Default: 'on'

ReferenceSignal

Reference signal to track. Must be a MATLAB `timeseries` object with real finite data points.

Default: [1x1 timeseries]

RelTol

Relative tolerance used to determine bounds as the signal approaches the reference signal. The bounds on the reference signal are given by:

$$y_u = (1 + RelTol)y_r + AbsTol$$

$$y_l = (1 - RelTol)y_r - AbsTol$$

Default: 0

RobustCost

Enable or disable robust treatment of outliers when evaluating the requirement. The software uses a Huber loss function to handle the outliers in the cost function and improves the fit quality. This option reduces the influence of outliers on the estimation without you manually modifying your data.

Must be one of the following:

- 'on' — When you call the `evalRequirement` method, the software uses a Huber loss function to evaluate the cost for the tracking error outliers. The tracking error is calculated as $e(t) = y_{ref}(t) - y_{test}(t)$. The software uses the error statistics to identify the outliers.

The exact cost function used, $F(x)$, depends on the requirement evaluation `Method`.

Method Name	Cost Function for Nonoutliers	Cost Function for Outliers
'SSE'	$F(x) = \sum_{t \in NOL} e(t) \times e(t)$ <p><i>NOL</i> is the set of nonoutlier samples.</p>	$F(x) = \sum_{t \in OL} w \times e(t) $ <p><i>w</i> is a linear weight. <i>OL</i> is the set of outlier samples.</p>
'SAE'	$F(x) = \sum_{t \in NOL} e(t) $ <p><i>NOL</i> is the set of nonoutlier samples.</p>	$F(x) = \sum_{t \in OL} w$ <p><i>w</i> is a constant value. <i>OL</i> is the set of outlier samples.</p>
'Residuals'	<p>The software does not remove the outliers.</p> $F(x) = \begin{bmatrix} e(0) \\ \vdots \\ e(N) \end{bmatrix}$ <p><i>N</i> is the number of samples.</p>	

- 'off'

Default: 'off'

Type

Tracking requirement type. Must be one of the following strings:

- '==' — Tracking objective.

sdo.requirements.SignalTracking

'<=' — Upper bound

- '>=' — Lower bound

Default: '=='

Weights

Weights to use when evaluating the tracking error between the reference and testpoint signals. Use weights to increase or decrease the significance of different time points.

Must be real finite positive vector with the same number of elements as the `Time` property of the MATLAB `timeseries` object in the `ReferenceSignal` property.

Methods

`evalRequirement` Evaluate tracking requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see [Copying Objects](#) in the MATLAB documentation.

Examples

Construct a signal tracking object and specify a reference signal.

```
r = sdo.requirements.SignalTracking;  
r.ReferenceSignal = timeseries(1-exp(-(0:10)'));
```

Alternatively, you can specify the reference signal during construction.

```
r = sdo.requirements.SignalTracking(...  
    'ReferenceSignal',timeseries(1-exp(-(0:10)')));
```

Alternatives

Use `getbounds` to get the bounds specified in a `Check Against Reference` block.

See Also

`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

sdo.requirements.SingularValue

Purpose

Singular value bound

Description

Specify frequency-dependent piecewise-linear upper and lower bounds on the singular values of a linear system. You can then optimize the model response to meet these bounds using `sdo.optimize` to .

You can specify upper or lower bounds, include multiple edges, and extend them to + or -infinity.

You must have Simulink Control Design software to specify singular value requirements.

Construction

`singval_req = sdo.requirements.SingularValue` creates a `sdo.requirements.SingularValue` object and assigns default values to its properties.

`singval_req = sdo.requirements.SingularValue(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Name-Value Pair Arguments

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

BoundFrequencies

Frequency values for the gain bound.

Specify the start and end frequencies for all the edges in the piecewise-linear bound. The property must be a `n x 2` array of finite doubles, where each row specifies the start and end frequencies of an edge in the piecewise-linear bound. The start

and end frequencies must define a positive length. The number of rows must match the number of rows of the `BoundMagnitudes` property.

Use `set` to set this and the `BoundMagnitudes` properties simultaneously.

Use the `FrequencyUnits` property to specify the frequency units.

Default: [1 10]

BoundMagnitudes

Magnitude values for the gain bound.

Specify the start and end gain values for all the edges in the piecewise-linear bound. The property must be a `nx2` array of finite doubles where each row specifies the start and end gains of an edge in the piecewise-linear bound. The number of rows must match the number of rows of the `BoundFrequencies` property.

Use `set` to set this and the `BoundFrequencies` properties simultaneously.

Use the `MagnitudeUnits` property to specify the magnitude units.

Default: [0 0]

Description

Requirement description. Must be a string.

Default: ''

FrequencyScale

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points. Must be one of the following strings:

sdo.requirements.SingularValue

- 'linear'
- 'log'

For example, if bound edges are at frequencies f_1 and f_2 , and the bound is to be evaluated at f_3 , the edges are interpolated as a straight lines. The x-axis is either linear or logarithmic.

Default: 'log'

FrequencyUnits

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'
- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'
- 'rad/week'
- 'rad/month'
- 'rad/year'

- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'
- 'cycles/year'

Default: 'rad/s'

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive frequency direction.

Specify whether the first and last edge of the bound extends to `inf` and `+inf` respectively. Use to bound signals that extend beyond the frequency values specified by the `BoundFrequencies` property.

sdo.requirements.SingularValue

Must be a 1x2 logical array of `true` or `false`. If `true`, the first or last edge of the piecewise linear bound is extended in the negative or positive direction.

Default: [0 0]

Type

Magnitude bound type. Must be:

- '`<=`' — Upper bound
- '`>=`' — Lower bound

Use to specify whether the piecewise-linear bound is an upper or lower bound. Use for upper bound and for lower bound.

Properties

BoundFrequencies

Frequency values for the gain bound.

Specify the start and end frequencies for all the edges in the piecewise-linear bound. The property must be a `nx2` array of finite doubles, where each row specifies the start and end frequencies of an edge in the piecewise-linear bound. The start and end frequencies must define a positive length. The number of rows must match the number of rows of the `BoundMagnitudes` property.

Use `set` to set this and the `BoundMagnitudes` properties simultaneously.

Use the `FrequencyUnits` property to specify the frequency units.

Default: [1 10]

BoundMagnitudes

Magnitude values for the gain bound.

Specify the start and end gain values for all the edges in the piecewise-linear bound. The property must be a nx2 array of finite doubles where each row specifies the start and end gains of an edge in the piecewise-linear bound. The number of rows must match the number of rows of the `BoundFrequencies` property.

Use `set` to set this and the `BoundFrequencies` properties simultaneously.

Use the `MagnitudeUnits` property to specify the magnitude units.

Default: [0 0]

Description

Requirement description. Must be a string.

Default: ''

FrequencyScale

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points. Must be one of the following strings:

- 'linear'
- 'log'

For example, if bound edges are at frequencies `f1` and `f2`, and the bound is to be evaluated at `f3`, the edges are interpolated as a straight lines. The x-axis is either linear or logarithmic.

Default: 'log'

FrequencyUnits

sdo.requirements.SingularValue

Frequency units of the requirement. Must be one of the following strings:

- 'Hz'
- 'rad/s'
- 'rpm'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rad/nanosecond'
- 'rad/microsecond'
- 'rad/millisecond'
- 'rad/minute'
- 'rad/hour'
- 'rad/day'
- 'rad/week'
- 'rad/month'
- 'rad/year'
- 'cycles/nanosecond'
- 'cycles/microsecond'
- 'cycles/millisecond'
- 'cycles/hour'
- 'cycles/day'
- 'cycles/week'
- 'cycles/month'
- 'cycles/year'

Default: 'rad/s'

MagnitudeUnits

Magnitude units of the requirement. Must be:

- 'db' (decibels)
- 'abs' (absolute units)

Default: 'db'

Name

Requirement name. Must be a string.

Default: ''

OpenEnd

Extend bound in a negative or positive frequency direction.

Specify whether the first and last edge of the bound extends to `inf` and `+inf` respectively. Use to bound signals that extend beyond the frequency values specified by the `BoundFrequencies` property.

Must be a 1x2 logical array of `true` or `false`. If `true`, the first or last edge of the piecewise linear bound is extended in the negative or positive direction.

Default: [0 0]

Type

Magnitude bound type. Must be:

- '<=' — Upper bound
- '>=' — Lower bound

sdo.requirements.SingularValue

Use to specify whether the piecewise-linear bound is an upper or lower bound. Use for upper bound and for lower bound.

Methods

`evalRequirement` Evaluate singular value bound on linear system

Copy Semantics

`Handle`. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a singular value object and specify bound frequencies and magnitudes.

```
r = sdo.requirements.SingularValue;  
set(r,'BoundFrequencies',[1 10; 10 100],...  
    'BoundMagnitudes',[1 1; 1 0]);
```

Alternatively, you can specify the frequency and magnitude during construction.

```
r = sdo.requirements.SingularValue(...  
    'BoundFrequencies', [1 10; 10 100], ...  
    'BoundMagnitudes', [1 1; 1 0]);
```

Alternatives

Use `getbounds` to get the bounds specified in a Check Singular Value Characteristics block.

See Also

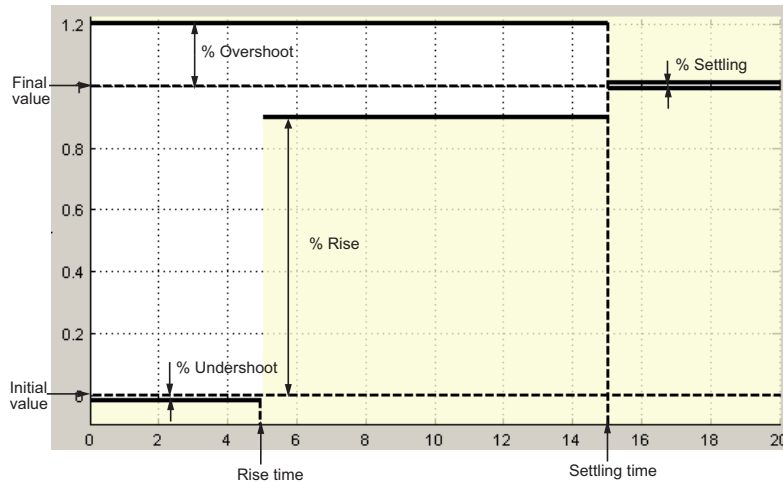
`copy` | `get` | `set`

How To

- Class Attributes
- Property Attributes

Purpose Step response bound on signal

Description Specify a step response envelope requirement on a time-domain signal. Step response characteristics such as rise-time and percentage overshoot define the step response envelope.



Construction

`step_req = sdo.requirements.StepResponseEnvelope` creates an `sdo.requirements.StepResponseEnvelope` object and assigns default values to its properties.

`step_req = sdo.requirements.StepResponseEnvelope(Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

sdo.requirements.StepResponseEnvelope

Input Arguments

Name-Value Pair Arguments

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

Description

Requirement description. Must be a string.

Default: ''

FinalValue

Final value of the step response. Must be a finite real scalar not equal to the `InitialValue` property.

Default: 1

InitialValue

Value of the signal level before the step response starts. Must be a finite real scalar not equal to the `FinalValue` property.

Default: 0

Name

Requirement name. Must be a string.

Default: ''

PercentOvershoot

The percentage amount by which the signal can exceed the final value before settling.

Must be a real finite scalar between [0 100] and greater than PercentSettling.

Use `set` to set this and the PercentSettling properties simultaneously.

Default: 10

PercentRise

The percentage of final value used with the RiseTime property to define the overall rise time characteristics.

Must be a real finite scalar between [0 100] and less than (100–PercentSettling).

Use `set` to set this and the PercentSettling properties simultaneously.

Default: 80

PercentSettling

The percentage of the final value that defines the settling range of settling time characteristic specified in the SettlingTime property.

Must be a real positive finite scalar between [0 100] and less than (100 – PercentRise) and less than PercentOvershoot.

Use `set` to set this and the PercentOvershoot and PercentRise properties simultaneously.

Default: 1

PercentUndershoot

The percentage amount by which the signal can undershoot the initial value.

Must be a positive finite scalar between [0 100].

sdo.requirements.StepResponseEnvelope

Default: 1

RiseTime

Time taken, in seconds, for the signal to reach a percentage of the final value specified in **PercentRise**.

Must be a finite positive real scalar and less than the **SettlingTime**. Time is relative to the **StepTime**.

Use **set** to set this and the **StepTime** and **SettlingTime** properties simultaneously.

Default: 5

SettlingTime

Time taken, in seconds, for the signal to settle within a specified range around the final value. This settling range is defined as the final value plus or minus the percentage of the final value, specified in **PercentSettling**.

Must be a finite positive real scalar, greater than **RiseTime**. Time is relative to the **StepTime**.

Use **set** to set this and the **RiseTime** properties simultaneously.

Default: 7

StepTime

Time, in seconds, when the step response starts.

Must be a finite real nonnegative scalar, less than the **RiseTime** property.

Use **set** to set this and the **RiseTime** properties simultaneously.

Default: 0

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Step response bound type.

This property is read-only and set to '<='.

Properties

Description

Requirement description. Must be a string.

Default: ''

FinalValue

Final value of the step response. Must be a finite real scalar not equal to the `InitialValue` property.

Default: 1

sdo.requirements.StepResponseEnvelope

InitialValue

Value of the signal level before the step response starts. Must be a finite real scalar not equal to the `FinalValue`. property.

Default: 0

Name

Requirement name. Must be a string.

Default: ''

PercentOvershoot

The percentage amount by which the signal can exceed the final value before settling.

Must be a real finite scalar between [0 100] and greater than `PercentSettling`.

Use `set` to set this and the `PercentSettling` properties simultaneously.

Default: 10

PercentRise

The percentage of final value used with the `RiseTime` property to define the overall rise time characteristics.

Must be a real finite scalar between [0 100] and less than $(100 - \text{PercentSettling})$.

Use `set` to set this and the `PercentSettling` properties simultaneously.

Default: 80

PercentSettling

The percentage of the final value that defines the settling range of settling time characteristic specified in the `SettlingTime` property.

Must be a real positive finite scalar between [0 100] and less than $(100 - \text{PercentRise})$ and less than `PercentOvershoot`.

Use `set` to set this and the `PercentOvershoot` and `PercentRise` properties simultaneously.

Default: 1

PercentUndershoot

The percentage amount by which the signal can undershoot the initial value.

Must be a positive finite scalar between [0 100].

Default: 1

RiseTime

Time taken, in seconds, for the signal to reach a percentage of the final value specified in `PercentRise`.

Must be a finite positive real scalar and less than the `SettlingTime`. Time is relative to the `StepTime`.

Use `set` to set this and the `StepTime` and `SettlingTime` properties simultaneously.

Default: 5

SettlingTime

Time taken, in seconds, for the signal to settle within a specified range around the final value. This settling range is defined as the final value plus or minus the percentage of the final value, specified in `PercentSettling`.

sdo.requirements.StepResponseEnvelope

Must be a finite positive real scalar, greater than RiseTime. Time is relative to the StepTime.

Use `set` to set this and the RiseTime properties simultaneously.

Default: 7

StepTime

Time, in seconds, when the step response starts.

Must be a finite real nonnegative scalar, less than the RiseTime property.

Use `set` to set this and the RiseTime properties simultaneously.

Default: 0

TimeUnits

Time units of the requirement. Must be one of the following strings:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Default: 'second'

Type

Step response bound type.

This property is read-only and set to '<='.

Methods

evalRequirement Evaluate step response bound

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a step response bound object and specify percent overshoot.

```
r = sdo.requirements.StepResponseEnvelope;  
r.PercentOvershoot = 20;
```

Alternatively, you can specify the percent overshoot during construction:

```
r = sdo.requirements.StepResponseEnvelope('PercentOvershoot',20);
```

Alternatives

Use getbounds to get the bounds specified in a Check Step Response Characteristics block.

See Also

copy | get | set

How To

- Class Attributes
- Property Attributes

sdo.SampledParameter

Superclasses Continuous

Purpose Sampled parameter

Syntax

```
p = sdo.SampledParameter(paramname)
p = sdo.SampledParameter(paramname,paramvalue)
p = sdo.SampledParameter(paramname,paramvalue,samplevalues)
```

Description A sampled parameter is a numeric parameter with a nominal value and set of sample values. The parameter can be scalar- or matrix-valued.

Typically, you use sampled parameters to create parametric models and evaluate model variations for robustness testing.

Construction `p = sdo.SampledParameter(paramname)` constructs a `sdo.SampledParameter` object for a parameter and assigns the specified name to the `Name` property and default values to the remaining properties.

`p = sdo.SampledParameter(paramname,paramvalue)` assigns the specified parameter value to the `Value` property.

`p = sdo.SampledParameter(paramname,paramvalue,samplevalues)` assigns the specified sample values to the `SampleValues` property.

Input Arguments

paramname

Parameter name, specified as a string inside single quotes (' ').

paramvalue

Scalar or matrix parameter value.

samplevalues

Scalar, matrix or cell array of parameter sample values.

Properties **Free**

Flag specifying whether the parameter is tunable or not.

Set the `Free` property to `true` (1) for tunable parameters and `false` (0) for parameters you do not want to tune (fixed).

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued parameters, you can:

- Fix individual matrix elements. For example `p.Free = [true false; false true]` or `p.Free([2 3]) = false`.
- Use scalar expansion to fix all matrix elements. For example `p.Free = false`.

Default: `true` (1)

Info

Structure array specifying parameter units and labels.

The structure has `Label` and `Unit` fields.

The array dimension must match the dimension of the `Value` property.

Use this property to store parameter units and labels that describe the parameter. For example `p.Info(1,1).Unit = 'N/m'`; or `p.Info(1,1).Label = 'spring constant'`.

Default: `''` for both `Label` and `Unit` fields

Maximum

Upper bound for the parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued parameters, you can:

- Specify upper bounds on individual matrix elements. For example `p.Maximum([1 4]) = 5`.

sdo.SampledParameter

- Use scalar expansion to set the upper bound for all matrix elements. For example `p.Maximum = 5`.

Default: `Inf`

Minimum

Lower bound for the parameter value.

The dimension of this property must match the dimension of the `Value` property.

For matrix-valued parameters, you can:

- Specify lower bounds on individual matrix elements. For example `p.Minimum([1 4]) = -5`.
- Use scalar expansion to set the lower bound for all matrix elements. For example `p.Minimum = -5`.

Default: `Inf`

Name

Parameter name.

This property is read-only and is set at object construction.

Default: `''`

SampleValues

Set of sample values for the parameter.

Must be a cell array of values. The elements of the cell array must have the same dimension as the `Value` property. If the `Value` property is a scalar, this property can be a vector.

Default: `[-1 1]`

Scale

Scaling factor used to normalize the parameter value.

The dimension of this property must match the dimension of the Value property.

For matrix-valued parameters, you can:

- Specify scaling for individual matrix elements. For example `p.Scale([1 4]) = 1`.
- Use scalar expansion to set the scaling for all matrix elements. For example `p.Scale = 1`.

Default: 1

Value

Scalar or matrix value of a parameter.

The dimension of this property is set at object construction.

Default: 0

Methods

Inherited Methods

`isreal`

Determine if parameter value, minimum and maximum are real

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify sample values during construction.

```
p = sdo.SampledParameter('K', eye(2), {0.9*eye(2) 1.1*eye(2)});
```

Construct a sampled parameter object and set its sample values.

sdo.SampledParameter

```
p = sdo.SampledParameter('K',eye(2));  
p.SampleValues = {0.9*eye(2) 1.1*eye(2)};
```

Alternatives “Optimize Parameters for Robustness (GUI)”

See Also param.Continuous | sdo.optimize

How To

- “Optimizing Parameters for Robustness”
- Class Attributes
- Property Attributes

Purpose

Parameter sampling options for `sdo.sample`

Description

Specify method options to generate parameter samples, using `sdo.sample`, for sensitivity analysis.

Construction

`opt = sdo.SampleOptions` creates an `sdo.SampleOptions` object and assigns default values to its properties.

Use dot notation to modify the property values. For example:

```
opt = sdo.SampleOptions;  
opt.Method = 'lhs';
```

Properties**Method**

Sampling method, specified as one of the following strings:

- 'random' — Random samples drawn from the probability distributions specified for the parameters.

Suppose you specified a value for the `RankCorrelation` property of the `sdo.ParameterSpace` object that you use for sampling. The software uses the Iman-Conover method to impose the parameter correlations.

- 'lhs' — Latin hypercube samples drawn from the probability distributions specified for the parameters. Use this option for a more systematic space-filling approach than random sampling.

Suppose you specified a value for the `RankCorrelation` property of the `sdo.ParameterSpace` object that you use for sampling. The software uses the Iman-Conover method to impose the parameter correlations.

Requires a Statistics Toolbox license.

- 'copula' — Random samples drawn from a copula. Use this option to impose correlations between the parameters. When you use this option, you must specify the value of the

sdo.SampleOptions

RankCorrelation property of the sdo.ParameterSpace object that you use for sampling.

Requires a Statistics Toolbox license.

For more information about the sampling methods, see “Sampling Parameters for Sensitivity Analysis”.

Default: 'random'

MethodOptions

Sample method options, applicable only when Method is 'copula', specified as a structure with the following fields:

- Family — Copula family, specified as one of the following strings:

- 'Gaussian' — Gaussian copula
- 't' — t copula

Default: 'Gaussian'

- Type — Rank correlation type, specified as one of the following strings

- 'Spearman' — Spearman's rank correlation
- 'Kendall' — Kendall's rank correlation

Default: 'Spearman'

- DOF — Degrees of freedom of t copula, specified as a positive integer.

For a Gaussian copula, specify DOF as []. Specification of DOF is required for a t copula.

Default: []

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify Random Sampling Method

```
opt = sdo.SampleOptions
```

```
opt =
```

```
SampleOptions with properties:
```

```
Method: 'random'  
MethodOptions: [0x0 struct]
```

Specify Latin Hypercube Sampling Method

```
opt = sdo.SampleOptions;  
sdo.Method = 'lhs';
```

Specify Copula-Based Sampling Method

```
opt = sdo.SampleOptions  
opt.Method = 'copula'  
opt.MethodOptions.Family = 't'  
opt.MethodOptions.DOF = 2
```

See Also

sdo.sample

Concepts

- Class Attributes
- Property Attributes
- “Sampling Parameters for Sensitivity Analysis”

sdo.SimulationTest

Purpose Simulation scenario description

Syntax `sim_obj = sdo.SimulationTest(modelname)`

Description Create a scenario to simulate a Simulink model. A simulation scenario specifies input signals, model parameter and initial state values, and signals to log for a model. Use a simulation scenario to simulate a model with alternative inputs and model parameter and initial state values, without modifying the model.

Construction `sim_obj = sdo.SimulationTest(modelname)` constructs an `sdo.SimulationTest` object and assigns the specified model name to the `ModelName` property and default values to the remaining properties.

You can also construct an `sdo.SimulationTest` object using the `sdo.Experiment.createSimulator` method of an `sdo.Experiment` object. The `createSimulator` method configures the properties of the `sdo.SimulationTest` object to simulate the model associated with the experiment.

Input Arguments

modelname

Simulink model name, specified as a string inside single quotes (' ').

The model must be on the MATLAB path.

Properties

InitialState

Model initial state for simulation.

This property can be any initial state format that `sim` command supports.

Inputs

Input signals.

Specify signals to apply to root level input ports when simulating the model. The signal can be any input signal format that the `sim` command supports.

Default: []

LoggedData

Data logged during simulation.

You must also specify the signals to log in the `LoggingInfo` property. The logged data is stored in a `Simulink.SimulationOutput` object and is populated by the `sim` method.

This property is read-only.

Default: []

LoggingInfo

Signals to log when simulating a model.

This property is a `Simulink.SimulationData.ModelLoggingInfo` object. Specify the signals to log in its `Signals` property.

Default: 1x1 `Simulink.SimulationData.ModelLoggingInfo` object

ModelName

Simulink model name associated with the simulation scenario. The model must be on the MATLAB path.

Name

Name of the scenario

Default: ''

Parameters

sdo.SimulationTest

Parameter values.

The software changes the model parameters to the specified values before simulating the model and restores them to their original value after the simulation completes.

This property must be a `param.Continuous` object.

Default: []

Methods

<code>find</code>	Find logged data set
<code>sim</code>	Simulate Simulink model using simulation scenario
<code>who</code>	List logged data names

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create Simulation Scenario for Model

Create a simulation scenario for a model and specify model signals to log.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;  
Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly';  
Pressures.OutputPortIndex = 1;  
simulator = sdo.SimulationTest('sdoHydraulicCylinder');  
simulator.LoggingInfo.Signals = [Pressures];
```

Create Simulation Scenario from Experiment

Specify an experiment.

```
experiment = sdo.Experiment('sdoRCCircuit');
```

Create a simulation scenario for the experiment.

```
sim_obj = createSimulator(experiment);
```

Alternatives

“Design Optimization to Meet Step Response Requirements (GUI)”

See Also

`sdo.optimize` | `sdo.Experiment.createSimulator` | `sdo.Experiment`

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Design Optimization to Meet a Custom Objective (Code)”
- “Estimate Model Parameter Values (Code)”
- “Estimate Model Parameters and Initial States (Code)”

How To

- Class Attributes
- Property Attributes

sdo.SimulationTest

Alphabetical List

copy

Purpose	Copy requirement
Syntax	<code>copy_req = copy(req)</code>
Description	<code>copy_req = copy(req)</code> copies a requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>) to a new object of the same type. For more information, see <code>copy</code> in the MATLAB documentation.
Input Arguments	req requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>)
Output Arguments	copy_req requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>), which is a copy of <code>req</code> .
See Also	<code>get</code> <code>handle</code>

sdo.requirements.BodeMagnitude.evalRequirement

Purpose	Evaluate Bode magnitude bound for linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluate whether a linear system satisfies the specified piecewise-linear Bode magnitude bound.
Input Arguments	<p>req</p> <p>sdo.requirements.BodeMagnitude object.</p> <p>For MIMO systems, the bound applies to each input/output (I/O) channel.</p> <p>lin_sys</p> <p>Linear system (tf, ss, zpk, frd, genss, or genfrd).</p>
Output Arguments	<p>c</p> <p>Column vector indicating the maximum signed distance of the system gain to each edge specified in <code>req</code>. Negative values indicate that the bound edge is satisfied and positive values that the bound edge is violated.</p> <p>For MIMO systems, a matrix of signed distances where each column represents an I/O pair and gives the distance of that IO pair gain to each edge in the bounds.</p>
Examples	<p>Evaluate Bode magnitude requirement.</p> <pre>req = sdo.requirements.BodeMagnitude; sys = tf(1,[1 2 2 1]); c = evalRequirement(req,sys);</pre> <p><code>c</code> is negative, which indicates that the system satisfies the gain requirement.</p>
See Also	sdo.requirements.BodeMagnitude get set copy

sdo.requirements.ClosedLoopPeakGain.evalRequirement

Purpose	Evaluate peak gain bound for linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluates whether a linear system satisfies the specified peak gain (infinity norm of the system) bound. The closed loop is computed using the feedback sign specified in the FeedbackSign property of req.
Input Arguments	req sdo.requirements.ClosedLoopPeakGain object. lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).
Output Arguments	c <ul style="list-style-type: none">• Signed distance of the closed-loop peak gain to the bound if the Type property of req is <= or ==. When <=, negative values indicate that the bound is satisfied while positive values indicate the bound is violated. When ==, any value other than 0 indicate that the bound is violated.• Peak gain if the Type property of req is min.
Examples	Evaluate peak gain requirement. <pre>req = sdo.requirements.ClosedLoopPeakGain; sys = tf(0.5,[1 3 3 1]); c = evalRequirement(req,sys);</pre> <p>c is negative, which indicates that the system satisfies the gain requirement.</p>
See Also	sdo.requirements.ClosedLoopPeakGain get set copy

sdo.requirements.GainPhaseMargin.evalRequirement

Purpose	Evaluate gain and phase margin bounds for linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluates whether a linear system satisfies the specified gain and phase margin bounds. The gain and phase margins are computed using the feedback sign specified in the <code>FeedbackSign</code> property of <code>req</code> .
Input Arguments	req sdo.requirements.GainPhaseMargin object. lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).
Output Arguments	c <ul style="list-style-type: none">• Signed distance of the computed gain and phase margins to the bound if the <code>Type</code> property of <code>req</code> is <code>>=</code> or <code>==</code>. Signed distance to the gain margin bound appear before the signed distance to the phase margin bound. Negative values indicate that the bound is satisfied while positive values indicate the bound is violated. Unstable loops return positive values. When <code>==</code>, any number other than 0 indicates that the bound is not satisfied.• Negative of the gain and phase margins such that minimizing the values maximizes the margins if the <code>Type</code> property of <code>req</code> is <code>'max'</code>. Unstable loops return positive values.
Examples	Evaluate gain and phase margin requirements. <pre>req = sdo.requirements.GainPhaseMargin; sys = tf(0.5,[1 3 3 1]); c = evalRequirement(req,sys);</pre>

sdo.requirements.GainPhaseMargin.evalRequirement

`c` is negative, which indicates that the system satisfies the gain and phase margin requirement.

get | set | copy

sdo.requirements.OpenLoopGainPhase.evalRequirement

Purpose	Evaluate gain and phase bounds on Nichols response of linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluates whether a linear system satisfies the specified open-loop gain and phase bounds on the Nichols response.
Input Arguments	req sdo.requirements.OpenLoopGainPhase object. lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).
Output Arguments	c Vector of maximum signed distances of the response to each piecewise linear edge. Negative values indicate that the bound edge is satisfied and positive values indicate the bound is violated.
Examples	Evaluate open-loop gain and phase requirements. <pre>req = sdo.requirements.OpenLoopGainPhase; sys = tf(0.5,[1 3 3 1]); c = evalRequirement(req,sys);</pre>
See Also	sdo.requirements.OpenLoopGainPhase get set copy

sdo.requirements.PZDampingRatio.evalRequirement

Purpose	Evaluate damping ratio bound on linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluates whether the poles of a linear system satisfies the specified damping ratio bound.
Input Arguments	req sdo.requirements.PZDampingRatio object. lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).
Output Arguments	c <ul style="list-style-type: none">• Signed distance of the damping ratio of each pole of the linear system to the bound, if the Type property of req is <code>>=</code>, <code><=</code> or <code>==</code>. Negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. When <code>==</code>, any value other than 0 indicates that the bound is violated.• Negative of the damping ratio such that minimizing the values maximizes the damping ratio, if the Type property of req is <code>'max'</code>.

Examples Evaluate damping ratio requirement.

```
req = sdo.requirements.PZDampingRatio;  
sys = tf(0.5,[1 3 3 1]);  
c = evalRequirement(req,sys);
```

`c` is negative, which indicates that the system satisfies the damping ratio requirement.
get | set | copy

sdo.requirements.PZNaturalFrequency.evalRequirement

Purpose	Evaluate natural frequency bound on linear system
Syntax	<code>c = evalRequirement(req, lin_sys)</code>
Description	<code>c = evalRequirement(req, lin_sys)</code> evaluates whether the poles of a linear system satisfies the specified natural frequency bound.
Input Arguments	<p>req</p> <p>Requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>).</p> <p>For MIMO systems, the bound applies to each input/output (I/O) channel.</p> <p>lin_sys</p> <p>Linear system (<code>tf, ss, zpk, frd, genss, or genfrd</code>).</p>
Output Arguments	<p>c</p> <ul style="list-style-type: none">• Signed distance of the natural frequency of each system pole to the bound. If the <code>Type</code> property of <code>req</code> is <code>>=</code>, <code><=</code>, negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. If <code>==</code>, any value other than 0 indicates that the bound is violated.• Negative of the natural frequency of the linear system poles such that minimizing the values maximizes the natural frequency, if the <code>Type</code> property of <code>req</code> is <code>'max'</code>.
Examples	<p>Evaluate natural frequency requirement.</p> <pre>req = sdo.requirements.PZNaturalFrequency; sys = tf(0.5,[1 3 3 1]); c = evalRequirement(req,sys);</pre> <p><code>c</code> is positive, which indicates that the system does not satisfy the natural frequency requirement.</p>

sdo.requirements.PZNaturalFrequency.evalRequirement

See Also

`sdo.requirements.PZNaturalFrequency` | `get` | `set` | `copy`

sdo.requirements.PZSettlingTime.evalRequirement

Purpose	Evaluate settling time bound on linear system
Syntax	<code>c = evalRequirement(req, lin_sys)</code>
Description	<code>c = evalRequirement(req, lin_sys)</code> evaluates whether the poles of a linear system satisfies the specified settling time bound.
Input Arguments	req sdo.requirements.PZSettlingTime object. lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).
Output Arguments	c <ul style="list-style-type: none">• Signed distance of the real component of each system pole to the bound, if the <code>Type</code> property of <code>req</code> is <code><=</code> or <code>==</code>. Negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. If <code>==</code>, values other than 0 indicate that the bound is violated.• Pole locations such that minimizing the values minimizes the settling time, if the <code>Type</code> property of <code>req</code> is <code>'min'</code>.
Examples	Evaluate settling time requirement. <pre>req = sdo.requirements.PZSettlingTime; sys = tf(0.5,[1 3 3 1]); c = evalRequirement(req,sys);</pre> <p><code>c</code> is positive, which indicates that the system does not satisfy the settling time requirement. get set copy</p>

sdo.requirements.SignalBound.evalRequirement

Purpose	Evaluate piecewise-linear bound
Syntax	<code>c = evalRequirement(req,sig)</code>
Description	<code>c = evalRequirement(req,sig)</code> evaluate whether a signal satisfies the specified piecewise-linear bounds.
Input Arguments	req sdo.requirements.SignalBound object. sig MATLAB timeseries object or nxm array, where the 1st column is time and subsequent columns are signal values.
Output Arguments	c Column vector indicating the maximum signed distance of the signal to each edge. Negative values indicate that the bound edge is satisfied and positive values indicate that the bound edge is violated. Matrix if multi-channeled signal.
Examples	Evaluate piecewise-linear bound on signal. <pre>req = sdo.requirements.SignalBound; sig = timeseries(1-exp(-(0:10)')); c = evalRequirement(req,sig);</pre> <p>c is negative, which indicates that the signal satisfies the bounds.</p>
See Also	get set copy

sdo.requirements.SignalTracking.evalRequirement

Purpose	Evaluate tracking requirement
Syntax	<pre>c = evalRequirement(req,sig) c = evalRequirement(req,sig,ref)</pre>
Description	<p><code>c = evalRequirement(req,sig)</code> evaluates whether a test point signal, <code>sig</code>, tracks the reference signal specified by a requirement object, <code>req</code>.</p> <p><code>c = evalRequirement(req,sig,ref)</code> evaluates whether <code>sig</code> tracks the reference signal specified by <code>ref</code>. <code>req</code> specifies the error computation options. Estimating parameters for multiple experiments requires you to repeatedly compare test point and reference signal sets. Use this syntax if you use the same evaluation criteria for all comparisons. You vary <code>sig</code> and <code>ref</code>, and re-use the requirement object, <code>req</code>.</p>
Input Arguments	<p>req sdo.requirements.SignalTracking object.</p> <p>sig MATLAB timeseries object or nxm array, where the 1st column is time and subsequent columns are signal values.</p> <p>ref Reference signal, specified as a MATLAB timeseries object.</p>
Output Arguments	<p>c</p> <ul style="list-style-type: none">• Measure of how well the test point signal matches the reference signal, if the <code>Type</code> property of <code>req</code> is <code>'=='</code>. Specify the algorithm used to compute the tracking measure through the <code>Method</code> property.• Signed distance of the test point signal to the reference signal, if the <code>Type</code> property of <code>req</code> is <code>'>='</code> or <code>'<='</code>. Negative values indicate the bound is satisfied while positive values indicate that the bound is violated.

sdo.requirements.SignalTracking.evalRequirement

The command compares the reference and test point signals only at time points that are in the range of both signals. Time points outside this range are ignored. The software uses the interpolation method specified by `ref.InterpolationTimes` to compare the data in the valid time range.

Examples

Evaluate Signal Tracking Requirement

Create the reference data.

```
time = (0:0.1:10)';  
data = 1-exp(-time);
```

Create the signal tracking requirement object. Specify the reference signal.

```
req = sdo.requirements.SignalTracking;  
req.ReferenceSignal = timeseries(data,time);
```

Obtain the test point signal.

```
sig = timeseries(1-exp(-time/2),time);
```

Evaluate the signal tracking requirement.

```
c = evalRequirement(req,sig);
```

Evaluate Tracking Using Requirement Object to Specify Error Computation Method

When you estimate parameters for multiple experiments, you repeatedly compare test point and reference signal sets. If you use the same evaluation criteria for all comparisons, you can use the `c = evalrequirement(req,sig,ref)` syntax. You vary `sig` and `ref`, and re-use the requirement object, `req`. `req` specifies the estimation error computation options.

For this example, create a reference and test point signal. Then, use a requirement object to evaluate the requirement.

sdo.requirements.SignalTracking.evalRequirement

Create the reference signal.

```
time = (0:0.1:10)';  
data = 1-exp(-time);  
ref = timeseries(data,time);
```

Create the signal tracking requirement object. Specify the error computation method.

For this example, specify 'Residuals' as the algorithm for error computation.

```
req = sdo.requirements.SignalTracking;  
req.Method = 'Residuals';
```

Obtain the test point signal.

```
sig = timeseries(1-exp(-time/2),time);
```

Evaluate the signal tracking requirement.

```
c = evalRequirement(req,sig,ref);
```

See Also

[get](#) | [set](#) | [copy](#)

sdo.requirements.SingularValue.evalRequirement

Purpose	Evaluate singular value bound on linear system
Syntax	<code>c = evalRequirement(req,lin_sys)</code>
Description	<code>c = evalRequirement(req,lin_sys)</code> evaluates whether a linear system satisfies the specified singular values bound.
Input Arguments	<p>req sdo.requirements.SingularValue object.</p> <p>For MIMO systems, the bound applies to each input/output (I/O) channel.</p> <p>lin_sys Linear system (tf, ss, zpk, frd, genss, or genfrd).</p>
Output Arguments	<p>c Column vector indicating the maximum signed distance of the system gain to each edge specified in <code>req</code>. Negative values indicate that the bound edge is satisfied and positive values indicate that the bound edge is violated.</p> <p>For MIMO systems, a matrix of signed distances where each column represents an I/O pair and gives the distance of that IO pair gain to each edge in the bounds.</p>
Examples	<p>Evaluate singular value requirement.</p> <pre>req = sdo.requirements.SingularValue; sys = tf(1,[1 2 2 1]); c = evalRequirement(req,sys);</pre> <p><code>c</code> is negative, which indicates that the system satisfies the gain requirement.</p>
See Also	sdo.requirements.SingularValue get set copy

sdo.requirements.StepResponseEnvelope.evalRequirement

Purpose Evaluate step response bound

Syntax `c = evalRequirement(req,sig)`

Description `c = evalRequirement(req,sig)` evaluate whether a signal satisfies specified step response bounds.

Input Arguments

req

sdo.requirements.StepResponseEnvelope object.

sig

MATLAB `timeseries` object or `nxm` array, where the 1st column is time and subsequent columns are signal values.

Numeric or generalized linear time invariant (LTI) model, if you have Simulink Control Design software.

Output Arguments

c

Column vector indicating the maximum signed distance of the signal to each edge in the step response envelope.

Signed distances to upper bound edges appear before signed distances to lower bounds edges. Negative values indicate that the bound edge is satisfied and positive values indicate that the bound edge is violated.

Examples

Evaluate step response bounds on signal.

```
req = sdo.requirements.StepResponseEnvelope;  
sig = timeseries(1-exp(-(0:10)'));  
c = evalRequirement(req,sig);
```

See Also

`get` | `set` | `copy`

get

Purpose	Get property values
Syntax	<code>get(req)</code> <code>get(req,PropertyName)</code>
Description	<code>get(req)</code> returns the value of all properties of the requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>). <code>get(req,PropertyName)</code> returns value of a specific property. Use a cell array of property names to return a cell array with multiple property values.
Input Arguments	req Requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>). PropertyName Name of the requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>) property.
Alternatives	“Getting Property Values”

Purpose	Set property values
Syntax	<code>set(req,Name,Value,)</code>
Description	<code>set(req,Name,Value,)</code> sets the property value of a requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>). Specify the property name and value using one or more <code>Name,Value</code> pair arguments.
Tips	<ul style="list-style-type: none">• Use <code>set</code> to simultaneously change properties that you cannot change independently.
Input Arguments	<p>req Requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>)</p> <p>Name,Value Property name of a requirement object (<code>sdo.requirements.StepResponseEnvelope, ...</code>), and the corresponding value to set.</p>
Examples	<p>Specify property values.</p> <pre>r = sdo.requirements.SignalBound; set(r,'BoundTimes',[0 5;5 10], ... 'BoundMagnitudes',[1.1 1.1; 1.01 1.01]);</pre>
Alternatives	“Setting Property Values”

getbounds

Purpose	Get bounds specified in Check block
Syntax	<code>bnds = getbounds(blockpath)</code>
Description	<code>bnds = getbounds(blockpath)</code> returns the bounds specified in the Check block specified by <code>blockpath</code> .
Input Arguments	blockpath Check block to get bounds from, specified as a full block path inside single quotes (' '). A block path is of the form <i>model/subsystem/block</i> that uniquely identifies a block in the model. The Simulink model must be open.
Output Arguments	bnds Cell array. The number of elements and objects they contain depends on the Check block type. <ul style="list-style-type: none">• Check Step Response Characteristics: Cell array of one element that contains a <code>sdo.requirements.StepResponseEnvelope</code> object.• Check Custom Bounds: Cell array of two elements — the first and second elements contain the following upper and lower bound values, respectively. Both elements are <code>sdo.requirements.SignalBound</code> objects.• Check Against Reference: Cell array of one element that contains a <code>sdo.requirements.SignalTracking</code> object.

Note Programmatically changing the bound values in the object returned does not update them in the Block Parameters dialog box.

Examples Retrieve bounds from a Check Step Response Characteristics block.

```
sldo_model1_stepblk;  
allBlkReq = getbounds('sldo_model1_stepblk/Step Response');
```

Type `allBlkReq{1}` to view the cell array element.

```
ans =
```

```
sdo.requirements.StepResponseEnvelope handle  
Package: sdo.requirements
```

```
Properties:
```

```
    InitialValue: 0  
    FinalValue: 1  
    StepTime: 0  
    RiseTime: 5  
    PercentRise: 80  
    SettlingTime: 7  
    PercentSettling: 1.0000  
    PercentOvershoot: 10.0000  
    PercentUndershoot: 1  
    Type: '<='  
    Name: ''  
    Description: ''  
    TimeUnits: 'seconds'
```

See Also

[sdo.optimize](#) | [Check Against Reference](#) | [Check Custom Bounds](#) | [Check Step Response Characteristics](#)

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”

sdo.requirements.PZDampingRatio.getOvershoot

Purpose Convert damping ratio to equivalent overshoot value

Syntax `overshoot = getOvershoot(req)`

Description `overshoot = getOvershoot(req)` converts the damping ratio value specified in the `DampingRatio` property of an `sdo.requirements.PZDampingRatio` object to an equivalent approximate second-order overshoot value.

Input Arguments

req
sdo.requirements.PZDampingRatio object.

Output Arguments

overshoot
Approximate second-order percent overshoot value, equivalent to the damping ratio value in `DampingRatio` property of `sdo.requirements.PZDampingRatio`.

Examples Convert damping ratio to approximate second-order overshoot value.

```
r = sdo.requirements.PZDampingRatio;  
r.DampingRatio = 0.1;  
overshoot = getOvershoot(r);
```

See Also `sdo.requirements.PZDampingRatio` | `evalRequirement` | `setOvershoot`

sdo.requirements.PZDampingRatio.setOvershoot

Purpose	Set overshoot to an equivalent damping ratio
Syntax	<code>req1 = setOvershoot(req,percent_overshoot)</code>
Description	<code>req1 = setOvershoot(req,percent_overshoot)</code> sets the damping ratio value to a value equivalent to percent overshoot.
Input Arguments	req sdo.requirements.PZDampingRatio object. percent_overshoot Percent overshoot value to compute damping ratio.
Output Arguments	req1 sdo.requirements.PZDampingRatio object whose DampingRatio property is the damping ratio value equivalent to percent_overshoot.
Examples	Specify overshoot bound. <code>req = sdo.requirements.PZDampingRatio setOvershoot(req,20)</code>
See Also	sdo.requirements.PZDampingRatio evalRequirement getOvershoot

makedist

Purpose Create probability distribution object

Syntax
`pd = makedist(distname)`
`pd = makedist(distname,Name,Value)`

Description `pd = makedist(distname)` creates a probability distribution object for the distribution `distname`, using the default parameter values.

Use `makedist` to specify normal or uniform distribution objects. If you have a Statistics Toolbox license, you can use `makedist` to create objects for other distributions, such as the Gamma or Weibull distributions. For more information, see `makedist` in the Statistics Toolbox documentation.

`pd = makedist(distname,Name,Value)` creates a probability distribution object with one or more distribution parameter values specified by name-value pair arguments.

Input Arguments

distname - Distribution name

string

Distribution name, specified as one of the following strings. The distribution specified by `distname` determines the class type of the returned probability distribution object.

Distribution Name	Description	Distribution Class
'Normal'	Normal distribution	prob.NormalDistribution
'Uniform'	Uniform distribution	prob.UniformDistribution

Name-Value Pair Arguments

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

Example: `makedist('Normal','mu',10)` specifies a normal distribution with parameter `mu` equal to 10, and parameter `sigma` equal to the default value of 1.

Normal Distribution

'mu' - Mean

0 (default) | scalar value

Example: `'mu',2`

Data Types

single | double

'sigma' - Standard deviation

1 (default) | nonnegative scalar value

Example: `'sigma',2`

Data Types

single | double

Uniform Distribution

'lower' - Lower parameter

0 (default) | scalar value

Example: `'lower',-4`

Data Types

single | double

'upper' - Upper parameter

1 (default) | scalar value greater than lower

Example: `'upper',2`

Data Types

single | double

makedist

Output Arguments

pd - Probability distribution

probability distribution object

Probability distribution, returned as a probability distribution object of the type specified by `distname`.

Examples

Create a Normal Distribution Object

Create a normal distribution object using the default parameter values.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Specify Parameters for a Normal Distribution Object

Create a normal distribution object with a mean value of `mu = 75`, and a standard deviation of `sigma = 10`.

```
pd = makedist('Normal','mu',75,'sigma',10)
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75
```

```
sigma = 10
```

See Also `sdo.ParameterSpace`

Purpose

Analyze how model parameters influence cost function

Syntax

```
r = sdo.analyze(x,y)
r = sdo.analyze(x,y,opts)
```

Description

`r = sdo.analyze(x,y)` returns an N_p -by- N_c table containing the pairwise linear correlation coefficients between each pair of columns in the `x` and `y` tables. `x` contains N_s samples of N_p model parameters. `y` contains N_s rows, each row corresponds to the cost function evaluation for a sample in `x`. Each column in `y` corresponds to a cost or constraint.

`r = sdo.analyze(x,y,opts)` specifies the analysis method(s) and method options using `opts`, an `sdo.AnalyzeOptions` object. If you specify multiple analysis methods, `r` is returned as a structure with fields for the results of each specified analysis method and method option combination.

Input Arguments

x - Model parameter samples

table

Model parameter samples, specified as an N_s -by- N_p table.

N_s is the number of samples, and N_p is the number of model parameters.

Generally, you use `sdo.sample` to generate `x`.

y - Cost function evaluations

table

Cost function and constraint evaluations for each sample in `x`, specified as an N_s -by- N_c table.

N_s is the number of samples, and N_c is the number of cost and constraints returned by the cost function.

Generally, you use `sdo.evaluate` to generate `y`.

opts - Analysis options

`sdo.AnalyzeOptions` object

Analysis options, specified as an `sdo.AnalyzeOptions` object.

Output Arguments

r - Analysis results

table | structure

Analysis results, returned as a table (when you specify a single analysis method) or a structure with table fields (when you specify multiple analysis methods).

Each table returned by `r` is an N_p -by- N_c table. N_p is the number of parameters, and N_c is the number of cost and constraints returned by the cost function.

Examples

Analyze Influence of Model Parameters on Cost Function

Create arbitrary `param.Continuous` objects.

```
p1 = param.Continuous('x1',1);  
p2 = param.Continuous('x2',1500);  
p = [p1;p2];
```

Specify the parameter space definition for the parameters.

```
ps = sdo.ParameterSpace(p);
```

Sample the parameters.

```
x = sdo.sample(ps,50);
```

Evaluate the cost function for the samples.

```
y = sdo.evaluate(@(p) sdoExampleCostFunction(p),ps,x);
```

Obtain the pairwise linear correlation coefficients for the parameters and the cost function.

```
r = sdo.analyze(x,y);
```

Specify Analysis Options

Create arbitrary param.Continuous objects.

```
p1 = param.Continuous('x1',1);  
p2 = param.Continuous('x2',1500);  
p = [p1;p2];
```

Specify the parameter space definition for the parameters.

```
ps = sdo.ParameterSpace(p);
```

Sample the parameters.

```
x = sdo.sample(ps,50);
```

Evaluate the cost function for the samples.

```
y = sdo.evaluate(@(p) sdoExampleCostFunction(p),ps,x);
```

Create an options object to use all available analysis methods.

```
opt = sdo.AnalyzeOptions;  
opt.Method = 'All';
```

Obtain the pairwise linear correlation coefficients for the parameters and the cost function.

```
r = sdo.analyze(x,y,opt);
```

r is a structure with table fields, with one table for each type of analysis specified by opt.

See Also

sdo.AnalyzeOptions | sdo.evaluate

Related Examples

- “Identify Key Parameters for Estimation (Code)”

Concepts

- “Sensitivity Analysis Methods”

sdo.evaluate

Purpose Evaluate cost function for samples

Syntax

```
[y,info] = sdo.evaluate(fcn,params)
[y,info] = sdo.evaluate(fcn,params,param_samples)
[y,info] = sdo.evaluate( ____,opts)
```

Description `[y,info] = sdo.evaluate(fcn,params)` evaluates the cost function, `fcn`, for samples of the parameter space specified by `params` (`sdo.ParameterSpace` object). The software generates a table of samples with $2Np+1$ rows and Np columns. These samples are generated per the distributions specified by the `ParameterDistributions`, `RankCorrelation`, and `Options` properties of `params`. The software evaluates the cost function for each row of the samples table. `y` is a table with one column for each cost or constraint output returned by `fcn` and $2Np+1$ rows. Np is the number of parameters specified in `params`.

`[y,info] = sdo.evaluate(fcn,params,param_samples)` evaluates the cost function for the specified parameter samples table, `param_samples`. For this syntax, you can specify `params` as an `sdo.ParameterSpace` object or a vector of `param.Continuous` objects.

`y` is a table with one column for each cost or constraint output returned by `fcn`. `y` contains as many rows as `param_samples`.

`[y,info] = sdo.evaluate(____,opts)` specifies evaluation options that configure the evaluation error handling, display, and parallel computing options. This syntax can include any of the input argument combinations in the previous syntaxes.

Input Arguments **fcn - Function to be minimized by sdo.optimize**
function handle

Function to be minimized by `sdo.optimize`, specified as a function handle.

For information about this function, see the description of the `opt_fcn` input argument in `sdo.optimize`. Also, see “Writing a Cost Function”.

params - Model parameters and states

sdo.ParameterSpace object | vector of param.Continuous objects

Model parameters and states, specified as an sdo.ParameterSpace object or a vector of param.Continuous objects.

If you specify `params` as a vector of param.Continuous objects, you must also specify `param_samples`.

param_samples - Parameter samples

table

Parameter samples, specified as a table.

`param_samples` contains columns that correspond to free scalar parameters and rows that are samples of these parameters. *Free scalar parameters* refers to all the parameters specified by `params` whose `Free` property is set to 1. Specifying this property value as 1 indicates that the software can vary the value of this parameter during optimization.

Each column name must be equal to the name of the corresponding scalar parameter.

opts - Evaluation options

sdo.EvaluateOptions object

Evaluation options, specified as an sdo.EvaluateOptions object.

Output Arguments**y - Cost function evaluation**

table

Cost function and constraint evaluations, returned as a table.

`y` is a table with one column for each cost or constraint output returned by `fcn`, and N_s rows.

If you specify `param_samples`, N_s is equal to the number of rows of `param_samples`. Otherwise, N_s is equal to $2N_p+1$. N_p is the number of parameters specified in `params`.

info - Evaluation information

structure

Evaluation information, returned as a structure with the following fields:

- **Status** — Evaluation status for each sample, returned as a cell array of strings.

Each entry of the cell array is one of the following strings:

- 'success' — Model evaluation was successful
- 'failure' — Model evaluation resulted all NaN results
- 'error' — Model evaluation resulted in an error

- **Stats** — Time to evaluate all samples, returned as a structure with the following fields:

- **StartTime** — Evaluation start time, returned as a six-element date vector containing the current date and time in decimal form: [year month day hour minute seconds]
- **EndTime** — Evaluation end time, returned as a six-element date vector containing the current date and time in decimal form: [year month day hour minute seconds]

To determine the total evaluation time, use `etime(info.EndTime,info.StartTime)`.

Examples

Evaluate Cost Function Value for Parameter Samples

Create an arbitrary `param.Continuous` object.

```
p = param.Continuous('x',1);
```

Specify the parameter space definition for the model parameter.

```
ps = sdo.ParameterSpace(p);
```

Evaluate the cost function.

```
[y,info] = sdo.evaluate(@(p) sdoExampleCostFunction(p),ps);
```

The software generates $3(2N_p+1)$, $N_p = 1$ parameter) samples and evaluates the `sdoExampleCostFunction` cost function for each sample.

See Also

`sdo.EvaluateOptions` | `sdo.optimize` | `sdo.sample` | `sdo.analyze` | `sdo.ParameterSpace`

Related Examples

- “Design Exploration using Parameter Sampling (Code)”
- “Identify Key Parameters for Estimation (Code)”

sdo.Experiment.createSimulator

Purpose Create simulation object from experiment to compare measured and simulated data

Syntax
`sim_obj = createSimulator(experiment)`
`sim_obj = createSimulator(experiment,sim_obj0)`

Description `sim_obj = createSimulator(experiment)` creates a `sdo.SimulationTest` object to simulate a model using the parameters and inputs specified in an experiment. You compare the simulated and measured outputs. `sim_obj` specifies the model stop time as the end time of the longest running experiment output signal.

`sim_obj = createSimulator(experiment,sim_obj0)` updates the values of the Parameters, InitialStates, Input and LoggingInfo properties of the `sdo.SimulationTest` object, `sim_obj0`. It does so using the corresponding properties specified by `experiment`. `sim_obj0.ModelName` must be the same as `experiment.ModelName`. You use this syntax to avoid creating a simulation scenario object (`sdo.SimulationTest` object) repeatedly and, instead, modify an existing simulation scenario object.

Input Arguments

experiment

Experiment, specified as an `sdo.Experiment` object.

sim_obj0

Simulation scenario, specified as an `sdo.SimulationTest` object.

Typically, you use the `createSimulator` method of an experiment to create `sim_obj0`, which returns an appropriately configured simulation scenario. You can construct `sim_obj0` using the syntax `sim_obj0 = sdo.SimulationTest(modelname)`. However, if you do so, then `sim_obj0.ModelName` must be the same as `experiment.ModelName`.

Output Arguments

sim_obj

Simulation scenario, returned as an `sdo.SimulationTest` object.

The properties of `sim_obj` are configured to simulate the model associated with `experiment` using the parameters, initial states and inputs defined by `experiment`.

When you use the syntax `sim_obj = createSimulator(experiment,sim_obj0)`, `sim_obj` is the same object as `sim_obj0`. However, it contains the `Parameters`, `InitialStates`, and `Input` property values of `experiment`. The `LoggingInfo` property of `sim_obj` is extended to include any additional signals from `experiment.OutputData`.

Examples

Create Simulation Scenario for Experiment

Specify an experiment for a model.

```
experiment = sdo.Experiment('sdoRCCircuit');
```

Create a simulation scenario for the experiment.

```
sim_obj = createSimulator(experiment);
```

Update Simulation Scenario for Experiment

Specify an experiment and a model parameter value for the experiment.

```
experiment = sdo.Experiment('sdoRCCircuit');
```

```
open_system('sdoRCCircuit');  
p = sdo.getParameterFromModel('sdoRCCircuit','C1');  
p.Value = 1e-6;  
p.Free = false;
```

```
experiment.Parameters = p;
```

Create a simulation scenario for the experiment.

```
sim_obj = createSimulator(experiment);  
sim_obj.Parameters.Value
```

sdo.Experiment.createSimulator

```
ans =  
  
1.0000e-06
```

Modify the model parameter value for the experiment.

```
experiment.Parameters.Value = 2e-6;
```

Update the simulation scenario.

```
sim_obj = createSimulator(experiment,sim_obj);  
sim_obj.Parameters.Value
```

```
ans =  
  
2.0000e-06
```

The value of the model parameter associated with `sim_obj` is updated.

See Also `sdo.SimulationTest`

Related Examples

- “Estimate Model Parameter Values (Code)”
- “Estimate Model Parameters and Initial States (Code)”

sdo.Experiment.getValuesToEstimate

Purpose Get model initial states and parameters for estimation from experiment

Syntax `parameters = getValuesToEstimate(experiment)`

Description `parameters = getValuesToEstimate(experiment)` returns the model initial states and parameters of an experiment that you want to estimate.

When you estimate parameters for multiple experiments, `getValuesToEstimate` tags each parameter to track its corresponding experiment. To update the experiments with their corresponding estimated parameter values, use `setEstimatedValues`.

Input Arguments

experiment

Experiment, specified as an `sdo.Experiment` object.

To get the model initial states and parameters for multiple experiments, use a vector of `sdo.Experiment` objects.

To specify that you want to estimate the value of a model initial state or parameter for an experiment, set its `Free` property to `true`. For example, `experiment.InitialStates(1).Free = true`.

Output Arguments

parameters

Model initial states and parameters of an experiment that you want to estimate, returned as a vector of `param.Continuous` objects.

When `experiment` specifies multiple experiments, `getValuesToEstimate` tags each entry of `parameters` to track its corresponding experiment. To update the experiments with their corresponding estimated parameter values, use `setEstimatedValues`.

sdo.Experiment.getValuesToEstimate

Examples

Get Model Initial States and Parameters to Estimate from Experiment

Specify an experiment with a model initial state and parameter that you want to estimate.

```
open_system('sdoRCCircuit');

experiment = sdo.Experiment('sdoRCCircuit');

experiment.InitialStates = sdo.getStateFromModel('sdoRCCircuit','C1');
experiment.Parameters = sdo.getParameterFromModel('sdoRCCircuit','C1');
```

Get the model initial states and parameters that you want to estimate from the experiment.

```
val = getValuesToEstimate(experiment)

val(1,1) =

    Name: 'sdoRCCircuit/C1:sdoRCCircuit.C1.vc_0'
    Value: 0
    Minimum: -Inf
    Maximum: Inf
    Free: 1
    Scale: 1
    dxValue: 0
    dxFree: 1
    Info: [1x1 struct]

val(2,1) =

    Name: 'C1'
    Value: 1.0000e-03
    Minimum: -Inf
    Maximum: Inf
    Free: 1
```

```
Scale: 0.0020  
Info: [1x1 struct]
```

```
2x1 param.Continuous
```

```
lists of methods, superclasses
```

`val(1,1)`, the initial voltage of the model capacitor block, `C1`, is the initial state specified by `experiment` for estimation. Execute `class(val(1,1))` to see that `val(1,1)` is a `param.State` object, representing a model initial state.

`val(2,1)`, the capacitance of the `C1` block, is the model parameter specified by `experiment` for estimation.

See Also

`sdo.Experiment` | `setEstimatedValues`

Related Examples

- “Estimate Model Parameters using Multiple Experiments (Code)”
- “Estimate Model Parameters Per Experiment (Code)”

sdo.Experiment.setEstimatedValues

Purpose Update experiments with estimated model initial states and parameter values

Syntax `experiment = setEstimatedValue(experiment0,parameters)`

Description `experiment = setEstimatedValue(experiment0,parameters)` updates the experiment with the estimated model initial states and parameter values.

`setEstimatedValues` is used with the `getValuesToEstimate` method. You use `getValuesToEstimate` to obtain the parameters that you want to estimate from an experiment. When you estimate parameters for multiple experiments, `getValuesToEstimate` tags each parameter to track its corresponding experiment. You use `setEstimatedValues` to update the experiments with their corresponding estimated parameter values.

Input Arguments

experiment0

Experiment, specified as an `sdo.Experiment` object.

To specify multiple experiments, use a vector of `sdo.Experiment` objects.

parameters

Estimated model initial states and parameters for experiments, specified as a vector of `param.Continuous` objects.

You obtain estimated parameters using `sdo.optimize`.

Output Arguments

experiment

Updated experiment, returned as an `sdo.Experiment` object.

If `experiment0` is a vector of experiments, then `experiment` is a corresponding vector of updated `sdo.Experiment` objects.

`setEstimatedValues` updates the values of the parameters and initial states specified in each of the experiments in `experiment0` using the corresponding entry in `parameters`.

Examples

Update Experiment with Estimated Parameter Value

Specify an experiment.

```
experiment = sdo.Experiment('sdoRCCircuit');
```

Typically, you also specify measured input/output data for the experiment.

Specify a model parameter for estimation.

```
sdoRCCircuit;  
C1_parameter = sdo.getParameterFromModel('sdoRCCircuit','C1');  
C1_parameter.Value = 460e-6;
```

```
experiment.Parameters = C1_parameter;
```

C1_parameter is the capacitance parameter of the C1 block. The initial guess for its value is 460 μF .

Estimate the parameter value.

Typically, you use `sdo.optimize` to get the estimated parameter values for an experiment. For this example, directly change the value of the capacitance parameter.

```
C1_parameter.Value = 1e-6;
```

Update the experiment with the estimated parameter.

```
experiment = setEstimatedValues(experiment,C1_parameter);
```

Use `experiment.Parameters.Value` to verify that the capacitance parameter's value is updated.

See Also `sdo.Experiment.getValuesToEstimate`

Related Examples

- “Estimate Model Parameters using Multiple Experiments (Code)”
- “Estimate Model Parameters Per Experiment (Code)”

sdo.getModelDependencies

Purpose List of model path dependencies

Syntax `dirs = sdo.getModelDependencies(modelname)`

Description `dirs = sdo.getModelDependencies(modelname)` returns paths containing model dependencies required for response optimization using parallel computing. The model must be open.

`sdo.getModelDependencies` does not return a complete list of model dependency paths when the dependencies are undetectable.

- Tips**
- Modify `dirs` to include:
 - Path dependencies that `sdo.getModelDependencies` cannot detect such as paths that contain code for your model callbacks
 - Paths that the workers cannot access directly such as path dependencies on local drive
 - Use `dirs` to set the `ParallelPathDependencies` property of `sdo.OptimizeOptions` to use for parallel computing.

Input Arguments **modelname**

Simulink model name that uses the parameter, specified as a string inside single quotes (' ').

Output Arguments **dirs**

Cell array of paths that contain model dependencies.

The cell array is empty when the model does not have any dependencies or `sdo.getModelDependencies` does not detect any dependencies.

Examples List model path dependencies required for parallel computing.

Open Simulink model.

```
% Copy Simulink model and boiler library to temporary folder.  
pathToLib = boilerpressure_setup;
```



```
% Add folder to search path.
addpath(pathToLib);
% Open Simulink model.
boilerpressure_demo

Enable parallel computing.

opts = sdo.OptimizeOptions;
opts.UseParallel = 'always';

Get model dependency paths.

dirs = sdo.getModelDependencies('boilerpressure_demo');

Add model dependencies to use during parallel computing.

opts.ParallelPathDependencies = dirs;
```

```
Make local paths accessible to remote workers.

Open Simulink model.

% Copy Simulink model and boiler library to temporary folder.
pathToLib = boilerpressure_setup;
% Add folder to search path.
addpath(pathToLib);
% Open Simulink model.
boilerpressure_demo

Enable parallel computing.

opts = sdo.OptimizeOptions;
opts.UseParallel = 'always';

Get model dependency paths.

dirs = sdo.getModelDependencies('boilerpressure_demo');
```

sdo.getModelDependencies

The resulting path is on a local drive, C:/.

Replace C:/ with valid network path accessible to remote workers.

```
dirs = regexprep(dirs, 'C:/', '\\\hostname\C$\')
```

Add model dependencies to use during parallel computing.

```
opts.ParallelPathDependencies = dirs;
```

Add path to model path dependency list.

Open Simulink model.

```
% Copy Simulink model and boiler library to temporary folder.  
pathToLib = boilerpressure_setup;  
% Add folder to search path.  
addpath(pathToLib);  
% Open Simulink model.  
boilerpressure_demo
```

Enable parallel computing.

```
opts = sdo.OptimizeOptions;  
opts.UseParallel = 'always';
```

Get model dependency paths.

```
dirs = sdo.getModelDependencies('boilerpressure_demo');
```

Append an additional path.

```
dirs=vertcat(dirs, '\\hostname\C$\matlab\work')
```

Add model dependencies to use during parallel computing.

```
opts.ParallelPathDependencies = dirs;
```

Alternatives “How to Use Parallel Computing”

See Also `sdo.optimize`

Tutorials • Improving Optimization Performance Using Parallel Computing

How To • “Speedup Using Parallel Computing”
 • “Analyze Model Dependencies”

sdo.getParameterFromModel

Purpose Design variable for optimization

Syntax
`p_des = sdo.getParameterFromModel(modelname,paramname)`
`p_des = sdo.getParameterFromModel(modelname)`

Description `p_des = sdo.getParameterFromModel(modelname,paramname)` creates an object from a Simulink model parameter that you can tune to satisfy design requirements during optimization. The model must be open.

`p_des = sdo.getParameterFromModel(modelname)` creates model parameter objects for all the parameters in the model.

Input Arguments

modelname

Simulink model name that uses the parameter, specified as a string inside single quotes (' ').

paramname

Model parameter name, specified as a string inside single quotes (' ') for one parameter or a cell array of strings for multiple parameters.

Output Arguments

p_des

A param.Continuous object for one parameter or an array of objects for multiple parameters.

If paramname is not specified, then p_des contains all the parameters of the model.

The Value property of the object is set to the current value of the model parameter.

Examples

Get Model Parameter as Optimization Design Variable

```
modelname = 'sldo_model1_stepblk';  
paramname = 'Kp';
```

```
sldo_model1_stepblk;  
  
p_des = sdo.getParameterFromModel(modelname,paramname);
```

Get Multiple Model Parameters as Optimization Design Variables

```
modelname = 'sldo_model1_stepblk';  
paramname = {'Kp','Ki','Kd'};
```

```
sldo_model1_stepblk;  
  
p_des = sdo.getParameterFromModel(modelname,paramname);
```

Get All Model Parameters as Optimization Design Variables

```
modelname = 'sldo_model1_stepblk';  
  
sldo_model1_stepblk;  
  
p_des = sdo.getParameterFromModel(modelname);
```

Alternatives

“Specify Design Variables”

See Also

[sdo.optimize](#) | [sdo.setValueInModel](#)

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Estimate Model Parameter Values (Code)”

sdo.getStateFromModel

Purpose

Initial state for estimation from Simulink model

Syntax

```
s = sdo.getStateFromModel(modelname,blockpath)
s = sdo.getStateFromModel(modelname)
```

Description

`s = sdo.getStateFromModel(modelname,blockpath)` creates a state parameter object for the state of a specified block in a Simulink model. Use the state object to either specify the initial-state value in an experiment or estimate it.

`s = sdo.getStateFromModel(modelname)` creates state parameter objects for all the states in the model.

Input Arguments

modelname

Simulink model name, specified as a string inside single quotes (' '). The model must be open.

blockpath

Block path of the block containing the required state, specified as a string inside single quotes (' ').

To specify multiple blocks, use a cell array of block path strings.

Output Arguments

s

Model state, returned as a `param.State` object.

`s.Value` is the initial value of the state in the model.

When you use the syntax `s =`

`sdo.getStateFromModel(modelname,blockpath)`, `s` contains the state of the corresponding block.

If `blockpath` specifies multiple blocks, then `sdo.getStateFromModel` returns a vector of `param.State` objects.

Examples

Get States from Model

```
modelname = 'sdoAircraft';  
open_system(modelname);  
  
blockpath = {'sdoAircraft/Actuator Model', ...  
            'sdoAircraft/Controller/Proportional plus integral compensator'};  
  
s = sdo.getStateFromModel(modelname,blockpath);
```

Get All Model States

```
modelname = 'sdoAircraft';  
open_system(modelname);  
  
s = sdo.getStateFromModel(modelname);
```

s is a vector containing nine `param.State` objects, which represent all the states of the `sdoAircraft` model.

See Also

`sdo.Experiment` | `param.State`

Related Examples

- “Estimate Model Parameter Values (Code)”
- “Estimate Model Parameters and Initial States (Code)”

sdo.getValueFromModel

Purpose	Get design variable value from model
Syntax	<code>param_value = sdo.getValueFromModel(modelname,param_des)</code>
Description	<code>param_value = sdo.getValueFromModel(modelname,param_des)</code> gets the value of a design variable in a Simulink model. The model must be open.
Input Arguments	<p>modelname Simulink model name that uses the parameter, specified as a string inside single quotes (' ').</p> <p>param_des Design variables, specified as:</p> <ul style="list-style-type: none">• A string inside single quotes (' ') for one variable or a cell array of strings for multiple variables• A <code>param.Continuous</code> object for one variable or a vector of objects for multiple variables, created using <code>sdo.getParameterFromModel</code>
Output Arguments	<p>param_value Design variable value in the model. A cell array for multiple variable values.</p>
Examples	<p>Get current design variable value from a model.</p> <pre>sldo_model1_stepblk; p_value = sdo.getValueFromModel('sldo_model1_stepblk','Kp');</pre> <p>Alternatively, type</p> <pre>p_des = sdo.getParameterFromModel('sldo_model1_stepblk','Kp'); p_value = sdo.getValueFromModel('sldo_model1_stepblk',p_des);</pre>
See Also	<code>sdo.optimize</code>

Purpose

Scatter plot of samples

Syntax

```
sdo.scatterPlot(X,Y)
sdo.scatterPlot(X)
[H,AX,BigAX,P,PAx] = sdo.scatterPlot( ___ )
```

Description

`sdo.scatterPlot(X,Y)` creates a matrix of subaxes containing scatter plots of the columns of `X` against the columns of `Y`. If `X` is p -by- n and `Y` is p -by- m , then `sdo.scatterPlot` creates a matrix of n -by- m subaxes. `X` and `Y` must have the same number of rows.

`sdo.scatterPlot(X)` is the same as `sdo.scatterPlot(X,X)`, except that the subaxes along the diagonal are replaced with histogram plots of the data in the corresponding column of `X`. For example, the subaxes along the diagonal in the i th column is replaced by `hist(X(:,i))`.

`[H,AX,BigAX,P,PAx] = sdo.scatterPlot(___)` returns the handles to the graphic objects. Use these handles to customize the scatter plot. For example, you can specify titles for the subaxes.

Input Arguments

X - Sampled data

table

Sampled data, specified as a table.

Y - Cost function evaluation data

table

Cost function evaluation data, specified as a table.

Output Arguments

H - Line object handles

matrix

Line object handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific line object. The line objects are used to create the scatter plots.

AX - Subaxes handles

matrix

Subaxes handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific subaxes.

BigAX - Big axes handle

scalar

Big axes handle, returned as a scalar. This is a unique identifier, which you can use to query and modify properties of the big axes. **BigAX** is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command will center text with respect to the big axes.

P - Patch object handles

vector | []

Patch object handles, returned as a vector or []. If histogram plots are created, then **P** is returned as a vector of patch object handles for the histogram plots. These are unique identifiers, which you can use to query and modify the properties of a specific patch object. If no histogram plots are created, then **P** is returned as empty brackets.

PAX - Handle to invisible histogram axes

vector | []

Handle to invisible histogram axes, returned as a vector or []. If histogram plots are created, then **PAX** is returned as a vector of histogram axes handles. These are unique identifiers, which you can use to query and modify the properties of a specific axes, such as the axes scale. If no histogram plots are created, then **PAX** is returned as empty brackets.

Examples

Scatter Plot of Parameter Samples and Cost Function Evaluations

Generally, you use the `sdo.scatterPlot(X,Y)` syntax with `X` specifying the samples and `Y` specifying the cost function value for each sample.

Use the `sdo.evaluate` command to perform the cost function evaluation to generate `Y`. For this example, obtain 100 samples of the `Ac` and `K` parameters of the `sdoHydraulicCylinder` model. Calculate the cost function as a function of `Ac` and `K`. Create a scatter plot to see the sample and cost function values.

Load the `sdoHydraulicCylinder` model.

```
load_system('sdoHydraulicCylinder');
```

Generate 100 samples of the `Ac` and `K` parameters.

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

```
ps = sdo.ParameterSpace(p);
```

```
X = sdo.sample(ps,100);
```

The first operation obtains the `Ac` and `K` parameters as a vector, `p`. The second operation creates an `sdo.ParameterSpace` object, `ps`, that specifies the probability distributions of the parameter samples. The third operation generates 100 samples of each parameter, returned as a `Table`, `X`.

Calculate the cost function value table.

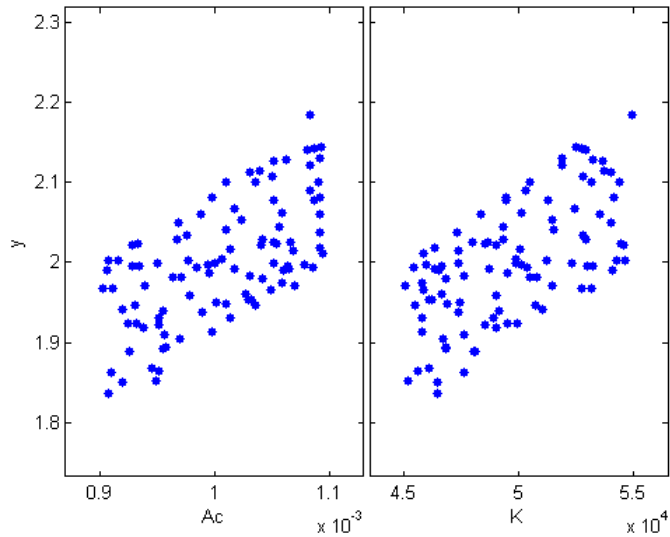
```
Ac_mean = mean(X(:,1));
```

```
K_mean = mean(X(:,2));
```

```
Y = table(X(:,1)/Ac_mean+X(:,2)/K_mean,'VariableNames',{'y'});
```

Create a scatter plot of `X` and `Y`.

```
sdo.scatterPlot(X,Y);
```



Scatter Plot of Parameter Samples

Sample the `Ac` and `K` parameters of the `sdoHydraulicCylinder` model. Use a scatter plot to analyze the samples.

Load the `sdoHydraulicCylinder` model.

```
load_system('sdoHydraulicCylinder');
```

Generate 100 samples of the `Ac` and `K` parameters.

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

```
ps = sdo.ParameterSpace(p);
```

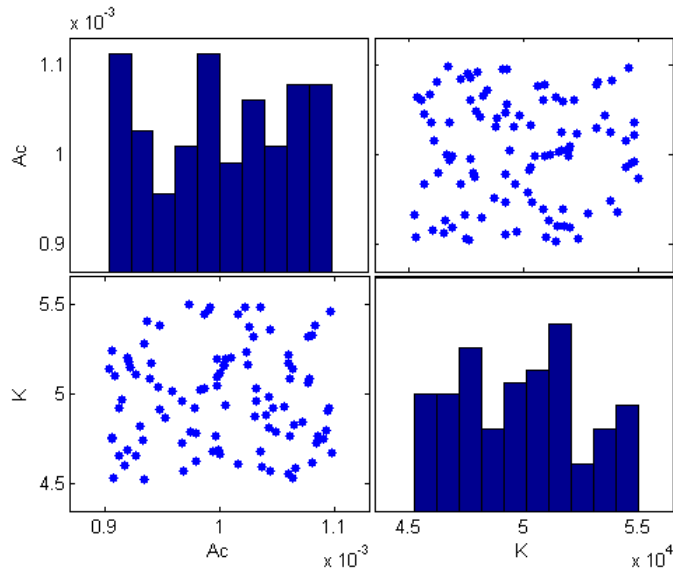
```
X = sdo.sample(ps,100);
```

The first operation obtains the `Ac` and `K` parameters as a vector, `p`. The second operation creates an `sdo.ParameterSpace` object, `ps`, that

specifies the probability distributions of the parameter samples. The third operation generates 100 samples of each parameter, returned as a Table, X.

Create a scatter plot of X.

```
sdo.scatterPlot(X);
```



Set Scatter Plot Properties Using Handles

Generate samples of the Ac and K parameters of the sdoHydraulicCylinder model.

```
load_system('sdoHydraulicCylinder');
```

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

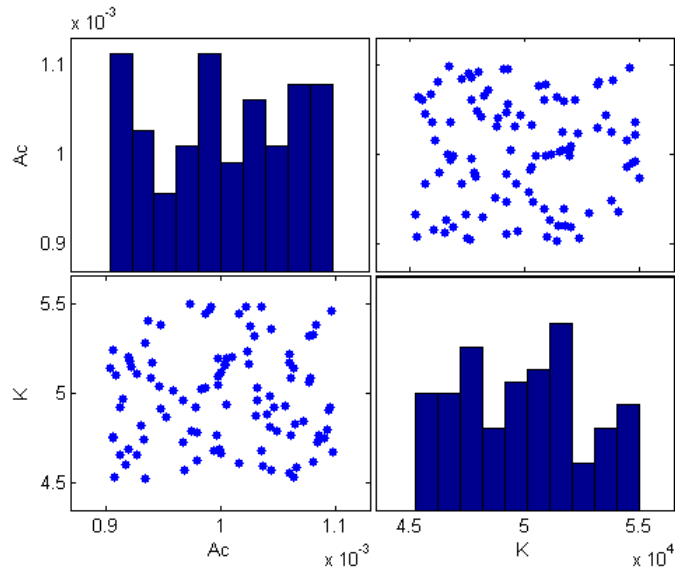
```
ps = sdo.ParameterSpace(p);
```

sdo.scatterPlot

```
X = sdo.sample(ps,100);
```

Create a scatter plot matrix and return the object handles and the axes handles.

```
figure  
[H,AX,BigAX,P,PAX] = sdo.scatterPlot(X);
```



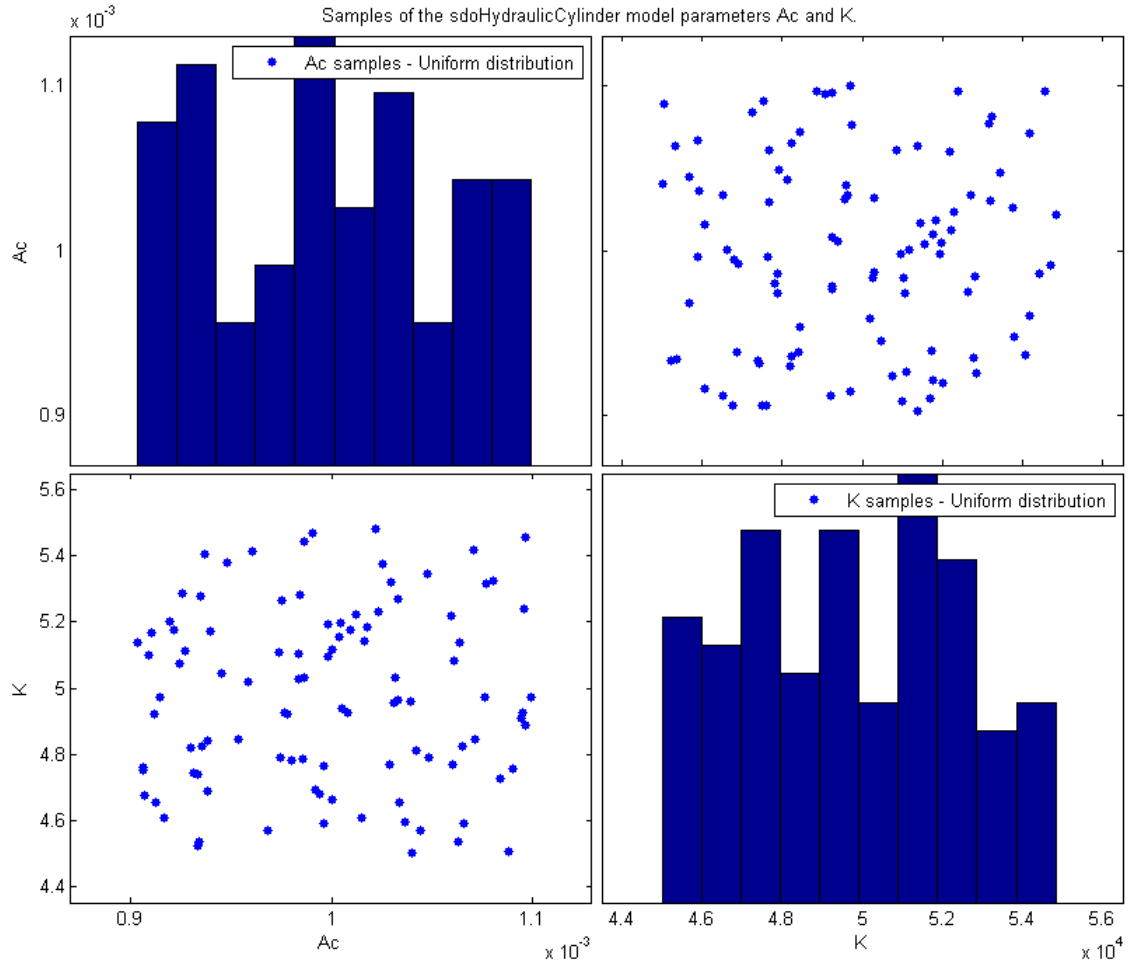
To set properties for the scatter plots, use the handles in H. To set properties for the histograms, use the patch handles in P. To set axes properties, use the axes handles, Ax, BigAX, and PAX.

Specify a title for the plot matrix and add legends specifying the sample distribution for each parameter.

```
title('Samples of the sdoHydraulicCylinder model parameters Ac and K.')
```

```
legend(AX(1),'Ac samples - Uniform distribution')
```

```
legend(AX(4),'K samples - Uniform distribution')
```



See Also `sdo.sample` | `sdo.evaluate`

sdo.scatterPlot

- Related** •
- Examples** •
- Concepts** •

Purpose	Determine if parameter value, minimum and maximum are real
Syntax	<code>isreal(param_obj)</code>
Description	<code>isreal(param_obj)</code> returns true (1) if the Value, Minimum and Maximum properties of <code>param_obj</code> are all real.
Input Arguments	param_obj A <code>param.Continuous</code> object.
Examples	<p>Determine if the value, maximum and minimum of a parameter are all real.</p> <pre>p= param.Continuous('K',eye(2)); isreal(p)</pre> <p>Because the Value, Minimum and Maximum properties of all parameters in <code>p</code> are all real, <code>isreal</code> returns 1.</p>
See Also	<code>param.Continuous</code>

sdo.ParameterSpace.addParameter

Purpose Add parameter to sdo.ParameterSpace object

Syntax
`ps = addParameter(ps0,p)`
`ps = addParameter(ps0,p,pdist)`

Description `ps = addParameter(ps0,p)` adds a model parameter, `p`, to an sdo.ParameterSpace object, `ps0`, and returns the updated object, `ps`. The software updates the `ParameterNames` property to include the parameter name.

The software also updates the `ParameterDistributions` property to specify the uniform distribution for the parameter. The software sets the values of the two parameters of the uniform distribution:

- **Lower** — Set to `p.Minimum`. If `p.Minimum` is equal to `-Inf`, then the software sets `Lower` to `0.9*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Lower` to `-1`.
- **Upper** — Set to `p.Maximum`. If `p.Maximum` is equal to `Inf`, then the software sets `Upper` to `1.1*p.Value`. Unless `p.Value` is equal to 0, in which case the software sets `Upper` to 1.

`ps = addParameter(ps0,p,pdist)` specifies the probability distribution of `p`.

Input Arguments

ps0
Parameter space, specified as an sdo.ParameterSpace object.

p
Model parameters and states, specified as a vector of param.Continuous objects.

For example,
`sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'})`.

pdist

Probability distribution of model parameters, specified as a vector of univariate probability distribution objects.

- If `pdist` is the same size as `p`, the software specifies each entry of `pdist` as the probability distribution of the corresponding parameter in `p`.
- If `pdist` contains only one distribution, the software specifies this object as the probability distribution for all the parameters in `p`.

Use the `makedist` command to create a univariate probability distribution object. For example, `makedist('Normal','mu',10,'sigma',3)`.

To check if `pdist` is a univariate distribution object, run `isa('pdist','prob.UnivariateDistribution')`.

Output Arguments

ps

Updated parameter space, returned as an `sdo.ParameterSpace` object.

Examples

Add Parameters to Parameter Space Object

Create an `sdo.ParameterSpace` object for the `Ac` parameter of the `sdoHydraulicCylinder` model.

```
load_system('sdoHydraulicCylinder');  
pAc = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac');  
ps = sdo.ParameterSpace(pAc);
```

Add the `K` parameter to `ps`.

```
pK = sdo.getParameterFromModel('sdoHydraulicCylinder','K');  
ps = addParameter(ps,pK);
```

sdo.ParameterSpace.addParameter

Add Parameter with Specified Distribution to Parameter Space Object

Create an `sdo.ParameterSpace` object for the `Ac` and `C1` parameters of the `sdoHydraulicCylinder` model.

```
load_system('sdoHydraulicCylinder');  
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','C1'});  
ps = sdo.ParameterSpace(p);
```

Add the `K` parameter to `ps`. Specify a normal distribution for `K`.

```
pK = sdo.getParameterFromModel('sdoHydraulicCylinder','K');  
pKdist = makedist('Normal','mu',pK.Value,'sigma',2);  
ps = addParameter(ps,pK,pKdist);
```

See Also

`sdo.sample` | `sdo.getParameterFromModel` | `makedist` |
`sdo.ParameterSpace.removeParameter`

Concepts

- “Sampling Parameters for Sensitivity Analysis”

sdo.ParameterSpace.removeParameter

Purpose	Remove parameter from sdo.ParameterSpace object
Syntax	<code>ps = removeParameter(ps0,p)</code>
Description	<code>ps = removeParameter(ps0,p)</code> removes the parameter, <code>p</code> , from the <code>sdo.ParameterSpace</code> object, <code>ps0</code> , and returns the updated object, <code>ps</code> .
Input Arguments	<p>ps0</p> <p>Parameter space, specified as an <code>sdo.ParameterSpace</code> object.</p> <p>p</p> <p>Parameters to be removed, specified as:</p> <ul style="list-style-type: none">• Vector of <code>param.Continuos</code> objects — Parameter objects. For example, <code>p = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac')</code>.• String — Parameter name. For example, <code>'Ac'</code>.
Output Arguments	<p>ps</p> <p>Updated parameter space, returned as an <code>sdo.ParameterSpace</code> object.</p>
Examples	<p>Remove Parameter from sdo.ParameterSpace Object</p> <p>Create an <code>sdo.ParameterSpace</code> object, <code>ps</code>, for the <code>Ac</code> and <code>K</code> parameters of the <code>sdoHydraulicCylinder</code> model.</p> <pre>load_system('sdoHydraulicCylinder'); p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'}); ps = sdo.ParameterSpace(p);</pre> <p>Remove <code>K</code> from <code>ps</code>.</p> <pre>ps = removeParameter(ps,p(2));</pre> <p>To verify that <code>ps</code> now contains only <code>Ac</code>, type <code>ps</code>.</p>

sdo.ParameterSpace.removeParameter

Now, remove Ac from ps using the parameter name.

```
ps = removeParameter(ps, 'Ac');
```

See Also

[sdo.ParameterSpace](#) | [sdo.ParameterSpace.addParameter](#) |
[sdo.getParameterFromModel](#)

sdo.ParameterSpace.setDistribution

Purpose	Set distribution of parameter in <code>sdo.ParameterSpace</code> object
Syntax	<code>ps = setDistribution(ps0,p,pdist)</code>
Description	<code>ps = setDistribution(ps0,p,pdist)</code> updates the <code>ParameterDistributions</code> property of the <code>sdo.ParameterSpace</code> object, <code>ps0</code> , for the specified parameters, <code>p</code> , and returns the updated object, <code>ps</code> .
Input Arguments	<p>ps0</p> <p>Parameter space, specified as an <code>sdo.ParameterSpace</code> object.</p> <p>p</p> <p>Parameters whose distributions are to be updated, specified as:</p> <ul style="list-style-type: none">• Vector of <code>param.Continuos</code> objects — Parameter objects. For example, <code>p = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac')</code>.• String — Parameter name. For example, <code>'Ac'</code>. <p>pdist</p> <p>Probability distribution for model parameters, specified as a vector of univariate probability distribution objects.</p> <ul style="list-style-type: none">• If <code>pdist</code> is the same size as <code>p</code>, the software specifies each entry of <code>pdist</code> as the probability distribution of the corresponding parameter in <code>p</code>.• If <code>pdist</code> contains only one distribution, the software specifies this object as the probability distribution for all the parameters in <code>p</code>. <p>Use the <code>makedist</code> command to create a univariate probability distribution object. For example, <code>makedist('Normal','mu',10,'sigma',3)</code>.</p>

sdo.ParameterSpace.setDistribution

To check if `pdist` is a univariate distribution object, run `isa('pdist','prob.UnivariateDistribution')`.

Output Arguments

ps

Updated parameter space, returned as an `sdo.ParameterSpace` object.

Examples

Set Distribution of Parameters in Parameter Space

Create an `sdo.ParameterSpace` object for the `Ac` and `K` parameters of the `sdoHydraulicCylinder` model.

```
load_system('sdoHydraulicCylinder');  
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});  
ps = sdo.ParameterSpace(p);
```

The call to `sdo.ParameterSpace` does not specify probability distributions for the parameters in `p`. So, by default, the software specifies the uniform distribution for all parameters in `p`.

```
ps.ParameterDistributions
```

```
ans =
```

```
1x2 UniformDistribution array
```

Specify the normal distribution for `Ac` and `K`.

```
pAcdist = makedist('Normal','mu',p(1).Value,'sigma',1);  
pKdist = makedist('Normal','mu',p(2).Value,'sigma',3);  
  
ps = setDistribution(ps,p,[pAcdist;pKdist]);
```

See Also

`sdo.sample` | `sdo.getParameterFromModel` | `makedist`

Purpose	Enable or disable all check blocks in model
Syntax	<code>chk_blk_state = sdo.setCheckBlockEnabled(modelname, state)</code>
Description	<p><code>chk_blk_state = sdo.setCheckBlockEnabled(modelname, state)</code> sets the Enabled parameter of all the check blocks in an open Simulink model to the specified value. The function returns the original value of the Enabled parameter of all the model check blocks.</p> <p>Use this function to disable the check blocks (model verification blocks) in a model before running an optimization for the model. After optimization completes, you can restore the enabled state of the model check blocks by calling this function again. Use the output from the previous call as the second input for the function.</p>
Input Arguments	<p>modelname Simulink model name, specified as a string inside single quotes (' '). The model must be open.</p> <p>state Switch enabling or disabling model check blocks, specified as either 'on' or 'off'. To restore the enabled state of the model check blocks, specify state as the output from the previous call to <code>sdo.setCheckBlockEnabled</code>.</p>
Output Arguments	<p>chk_blk_state Original values of the Enabled block parameter of the model check blocks, returned as a cell array of strings.</p>
Examples	<p>Disable Model Check Blocks</p> <p>Disable the model check blocks in a model.</p> <pre>modelname = 'pidtune_demo';</pre>

sdo.setCheckBlockEnabled

```
open_system(modelname);  
state = 'off';  
chkBlkState = sdo.setCheckBlockEnabled(modelname, state);
```

To restore the enabled state of the model check blocks, use:

```
sdo.setCheckBlockEnabled(modelname, chkBlkState)
```

Alternatives

You can open each model verification block in a model and select or clear the **Enable assertion** check box.

Purpose

Set design variable value in model

Syntax

```
sdo.setValueInModel(modelname,param_des)  
sdo.setValueInModel(modelname,param_des,value)
```

Description

`sdo.setValueInModel(modelname,param_des)` sets the value of a parameter in an open Simulink model to the Value property of the design variable `param_des`.

You generally use this command to update the Simulink model with optimized parameter values.

`sdo.setValueInModel(modelname,param_des,value)` sets the parameter to the value you specify.

Input Arguments

modelname

Simulink model name that uses the parameter, specified as a string inside single quotes (' ').

param_des

Design variable, specified as

- A `param.Continuous` object for one variable or a vector of objects for multiple variables, created using `sdo.getParameterFromModel`.
- A string inside single quotes (' ') for one variable or a cell array of strings for multiple variables.

You must also specify the `value` argument.

value

Value to set for the design variable.

Use a cell array with the same number of elements as the number of variables in `param_des` for setting values of multiple design variables.

sdo.setValueInModel

Examples

Change the design variable value in a model.

```
sldo_model1_stepblk;  
p_des = sdo.getParameterFromModel('sldo_model1_stepblk','Kp');  
p_des.Value = 1.1*p_des.Value;  
sdo.setValueInModel('sldo_model1_stepblk',p_des);
```

The value of Kp is set to the Value property of p_des.

Alternatives

“Update Model with Design Variables Set”

See Also

sdo.optimize

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Design Optimization to Track Reference Signal (GUI)”
- “Design Optimization Using Frequency-Domain Check Blocks (GUI)”

Purpose

Design optimization problem solution

Syntax

```
[param_opt,opt_info] = sdo.optimize(opt_fcn,param)
[param_opt,opt_info] = sdo.optimize(opt_fcn,param,options)
[param_opt,opt_info] = sdo.optimize(prob)
```

Description

[param_opt,opt_info] = sdo.optimize(opt_fcn,param) uses `fmincon` (the default optimization method) to solve a design optimization problem of the form:

$$\min_p F(p) \text{ subject to } \begin{cases} C_{leq}(p) \leq 0 \\ C_{eq}(p) = 0 \\ A \times p \leq B \\ A_{eq} \times p = B_{eq} \\ lb \leq p \leq ub \end{cases}$$

where

- p — Design variable
- C_{leq}, C_{eq} — Nonlinear inequality and equality constraints
- A, B — Linear inequality constraints
- A_{eq}, B_{eq} — Linear equality constraints
- lb, ub — Upper and lower bounds on p

[param_opt,opt_info] = sdo.optimize(opt_fcn,param,options) specifies the optimization options. For parameter estimation, you typically use the Nonlinear Least Squares method:

```
opts = sdo.OptimizeOptions('Method','lsqnonlin');
```

[param_opt,opt_info] = sdo.optimize(prob) uses a structure that contains the function to be minimized, design variables and optimization options.

Input Arguments

opt_fcn

Function to be minimized. The optimization solver calls this function during optimization.

The function requires:

- One input argument, which is a vector of `param.Continuous` objects to be tuned.

To pass additional input arguments, use an anonymous function. For example, `new_fcn = @(p) fcn(p, arg1, arg2, ...)`.

- One output argument, which is a structure with one or more of the following fields:
 - `F` — Value of the cost function evaluated at `p`. The solver minimizes `F`.
`F` is a 1x1 double.
 - `C1eq` — Value of the nonlinear inequality constraint violations evaluated at `p`. The solver satisfies `C1eq(p) <= 0`.
`C1eq` is a double `m`x1 vector, where `m` is the number of nonlinear inequality constraints.
 - `Ceq` — Value of the nonlinear equality constraint violations evaluated at `p`. The solver satisfies `Ceq(p) == 0`.
The value is a double `r`x1 vector, where `r` is the number of nonlinear equality constraints.
 - `1eq` — Value of the linear inequality constraint violations evaluated at `p`. The solver satisfies `1eq(p) <= 0`.
`1eq` is a double `n`x1 vector, where `n` is the number of linear inequality constraints.
 - `eq` — Value of the linear equality constraint violations evaluated at `p`. The solver satisfies `eq(p) == 0`.
`eq` is a double `s`x1 vector or `[]`, where `s` is the number of linear equality constraints.

To specify a pure feasibility problem, omit F or set $F = []$. To specify a minimization problem, omit C_{leq} , C_{eq} , leq and eq or set their values to $[]$.

The software computes gradients of the cost and constraint violations using numeric perturbation. If you want to specify how the gradients are computed, include a second output argument and set the `GradFcn` property of `sdo.OptimizeOptions` to 'on'. This argument must be a structure with one or more of the following fields:

- F — Double $n \times 1$ vector that contains $dF(p)/dp$, where n is the number of scalar parameters.
- C_{leq} — Double $n \times m$ matrix that contains $dC_{leq}(p)/dp$, where m is the number of nonlinear inequality constraints.
- C_{eq} — Double $n \times r$ matrix that contains $dC_{eq}(p)/dp$, where r is the number of nonlinear equality constraints.

For an example, type `edit sdoExampleCostFunction`.

param

A `param`. Continuous object or a vector of objects.

options

Optimization options.

`options` is an options set, created using `sdo.OptimizeOptions`. Use this options set to specify:

- Optimization method
- Maximum number of iterations
- Tolerances

prob

Structure with the following fields:

- **OptFcn** — Name of the function to be minimized. See `opt_fcn` for the input and output argument requirements of this function.
- **Parameters** — A `param.Continuous` object or a vector of objects
- **Options** — Optimization options, specified using `sdo.OptimizeOptions`

Output Arguments

param_opt

A `param.Continuous` object or vector of objects, containing the optimized parameter values in the `Value` property.

opt_info

Optimization information. Structure with one or more of the following fields:

- **F** — Optimized cost (objective) value.
- **C1eq** — Optimized nonlinear inequality constraint violations.

The field appears if you specify a nonlinear inequality constraint in `opt_fcn`.

The value is a `m`x1 vector, where the order of the elements correspond to the order specified in `opt_fcn`. Positive values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- **Ceq** — Optimized nonlinear equality constraint violations.

The field appears if you specify a nonlinear equality constraint in `opt_fcn`.

The value is a double `r`x1 vector, where the order of the elements correspond to the order specified in `opt_fcn`. Any nonzero values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- **1eq** — Optimized linear equality constraint violations.

The field appears if you specify a linear equality constraint in `opt_fcn`.

The value is a double $n \times 1$ vector, where the order of the elements correspond to the order specified in `opt_fcn`. Nonzero values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- `eq` — Optimized linear equality constraint violations.

The field appears if you specify linear equality constraints in `opt_fcn`.

The value is a double $s \times 1$ vector, where the order of the elements correspond to the order specified in `opt_fcn`. Nonzero values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- `Gradients` — Cost and constraint gradients at the optimized parameter values. See “How the Optimization Algorithm Formulates Minimization Problems” on how the solver computes gradients.

This field appears if the solver specified in the `Method` property of `sdo.OptimizeOptions` computes gradients.

The value is a structure whose fields are dependent on `opt_fcn`.

- `exitflag` — Integer identifying the reason the algorithm terminated. See `fmincon`, `patternsearch` and `fminsearch` for a list of the values and the corresponding termination reasons.
- `iterations` — Number of optimization iterations
- `SolverOutput` — A structure with solver-specific output information. The fields of this structure depends on the optimization solver specified in the `Method` property of `sdo.OptimizeOptions`. See `fmincon`, `patternsearch` and `fminsearch` for a list of solver outputs and their description.
- `Stats` — A structure that contains statistics collected during optimization, such as start and end times, number of function evaluations and restarts.

Examples

Optimize Model Response

Create design variables.

```
p = param.Continuous('x',1);
```

Specify optimization options.

```
opts = sdo.OptimizeOptions;  
opts.GradFcn = 'on';
```

Optimize the parameter.

```
[pOpt,opt_info] = sdo.optimize(@(p) sdoExampleCostFunction(p),p,opts);
```

Tips

- By default, the software displays the optimization information for each iteration in the MATLAB command window. To learn more about the information displayed, see:
 - “Iterative Display” when the optimization method is specified as 'fmincon' (default), 'fminsearch', or 'lsqnonlin'
 - “Display to Command Window Options” when the optimization method is specified as 'patternsearch'

You can configure the level of this display using the `MethodOptions.Display` property of an optimization option set.

Alternatives

“Design Optimization to Meet Step Response Requirements (GUI)”

|
.
.

See Also

`sdo.OptimizeOptions` | `function_handle` (@) | `param.Continuous`

Related Examples

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Estimate Model Parameter Values (Code)”
- “Design Optimization to Meet a Custom Objective (Code)”

Concepts

- “Writing a Cost Function”
- “Optimization Options”
- “Estimation Options”

sdo.sample

Purpose Generate parameter samples

Syntax
`x = sdo.sample(ps)`
`x = sdo.sample(ps,N)`
`x = sdo.sample(____,opt)`

Description `x = sdo.sample(ps)` generates samples using the specified parameter space definition, `ps`. The output sample table, `x`, has $2Np+1$ rows and Np columns. Each column corresponds to a parameter and each row corresponds to a sample of the parameters. Np is the number of parameters in `ps`. The samples are generated as per the `ParameterDistributions`, `RankCorrelation`, and `Options` property of `ps`.

`x = sdo.sample(ps,N)` specifies the number of samples to be generated. `x` is a table with N rows and Np columns.

`x = sdo.sample(____,opt)` specifies sampling options such as the sampling method. This syntax can include any of the input argument combinations in the previous syntaxes.

Input Arguments

ps - Parameter space distribution

`sdo.ParameterSpace` object

Parameter space distribution definition, specified as an `sdo.ParameterSpace` object.

N - Number of samples

positive integer

Number of samples to be generated for the parameters, specified as a positive integer.

Ideally, you want to use the smallest number of samples that yield useful results, because each sample requires a model evaluation.

As the number of parameters increases, the number of samples needed to explore the design space generally increases. For correlation or regression analysis, consider using $10Np$ samples, where Np is the number of parameters.

Example: 10

opt - Sampling options

sdo.SampleOptions object

Sampling options, specified as an sdo.SampleOptions object.

Output Arguments

x - Parameter samples

table

Parameter samples, returned as a table.

x has N_s rows and N_p columns. Each column corresponds to a parameter and each row corresponds to a sample of the parameters. N_p is the number of parameters in ps . If you specify N , N_s is equal to N . Otherwise, N_s is equal to $2N_p+1$.

Examples

Generate Parameter Samples

Generate samples for the A_c and K parameters of the sdoHydraulicCylinder model.

Open the model.

```
open_system('sdoHydraulicCylinder');
```

Obtain the parameters from the model.

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

Create an sdo.ParameterSpace object to specify the sample distributions.

```
ps = sdo.ParameterSpace(p);
```

Generate samples for the parameters.

```
x = sdo.sample(ps);
```

Specify Number of Samples

Generate 50 samples for the Ac and K parameters of the sdoHydraulicCylinder model.

Open the model.

```
open_system('sdoHydraulicCylinder');
```

Obtain the parameters from the model.

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

Create an sdo.ParameterSpace object to specify the sample distributions.

```
ps = sdo.ParameterSpace(p);
```

Generate 50 samples for the parameters.

```
x = sdo.sample(ps,50);
```

Specify Sampling Options

Open the model.

```
open_system('sdoHydraulicCylinder');
```

Obtain the parameters from the model.

```
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
```

Create an sdo.ParameterSpace object to specify the sample distributions.

```
ps = sdo.ParameterSpace(p);
```

Specify the sampling method used by the software.

```
opt = sdo.SampleOptions;  
opt.Method = 'lhs';
```

The 'lhs' (Latin hypercube) sampling method requires a Statistics Toolbox license.

Generate 50 samples for the parameters using Latin hypercube sampling.

```
x = sdo.sample(ps,50,opt);
```

See Also

[sdo.SampleOptions](#) | [sdo.evaluate](#)

Related Examples

- “Design Exploration using Parameter Sampling (Code)”
- “Identify Key Parameters for Estimation (Code)”

Concepts

- “Sampling Parameters for Sensitivity Analysis”

sdo.SimulationTest.find

Purpose Find logged data set

Syntax `data = find(sim_obj,data_name)`

Description `data = find(sim_obj,data_name)` searches for an element with a specific name in the `LoggedData` property of `sim_obj`. Use `who` to find possible names.

Input Arguments

sim_obj
sdo.SimulationTest object

data_name
Data set name to search for, specified as a string inside single quotes (' ').

Output Arguments

data
Logged simulation data for the data set name specified in `data_name`.

Examples Find logged data set.

Log model signals.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;  
Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly';  
Pressures.OutputPortIndex = 1;  
simulator = sdo.SimulationTest('sdoHydraulicCylinder');  
simulator.LoggingInfo.Signals = Pressures;
```

Run a simulation

```
sim = sim(simulator)
```

Search for logged data.

```
sim_log = find(simulator,'sdoHydraulicCylinder');
```


See Also

`sdo.optimize` | `sim` | `who`

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Design Optimization to Meet a Custom Objective (Code)”

sdo.SimulationTest.sim

Purpose Simulate Simulink model using simulation scenario

Syntax `sim_out = sim(sim_obj)`

Description `sim_out = sim(sim_obj)` simulates a Simulink model using a simulation scenario.

Tips

- Before simulating the model, specify the parameter values and signals to log in the `Parameters` and `LoggingInfo` properties of the `sim_obj`. The software restores the parameter values and logging settings to their original values after simulation.

Input Arguments

sim_obj
sdo.SimulationTest object

Output Arguments

sim_out
sdo.SimulationTest object which contains the logged data in the `LoggedData` property.

Examples Simulate a model and log model signal during simulation.

Log model signals.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;  
Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly';  
Pressures.OutputPortIndex = 1;  
simulator = sdo.SimulationTest('sdoHydraulicCylinder');  
simulator.LoggingInfo.Signals = [Pressures];
```

Specify parameter values.

```
Ac = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac');  
Ac.Value = 0.5;  
simulator.Parameters = Ac;
```

Simulate the model.

```
sim_obj = sim(simulator);
```

The specified signal Pressure is logged during simulation.

See Also

`sdo.optimize` | `find` | `who`

Tutorials

- “Design Optimization to Meet Step Response Requirements (Code)”
- “Design Optimization to Meet a Custom Objective (Code)”

sdo.SimulationTest.who

Purpose	List logged data names
Syntax	<code>names = who(sim_obj)</code>
Description	<code>names = who(sim_obj)</code> returns a list of logged data names.
Input Arguments	sim_obj sdo.SimulationTest object
Output Arguments	names Cell array of logged data set names.
Examples	<p>List logged data set names.</p> <p>Log model signals.</p> <pre>Pressures = Simulink.SimulationData.SignalLoggingInfo; Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly'; Pressures.OutputPortIndex = 1;</pre> <p>Store logged signal data.</p> <pre>simulator = sdo.SimulationTest('sdoHydraulicCylinder'); simulator.LoggingInfo.Signals = Pressures; simulator = sim(simulator);</pre> <p>Find logged data sets.</p> <pre>names = who(simulator);</pre>
See Also	<code>sdo.optimize</code> <code>find</code> <code>sim</code>
Tutorials	<ul style="list-style-type: none">• “Design Optimization to Meet Step Response Requirements (Code)”• “Design Optimization to Meet a Custom Objective (Code)”

Purpose	Open Design Optimization tool
Syntax	<code>sdotool(modelname)</code> <code>sdotool(sdosession)</code>
Description	<code>sdotool(modelname)</code> opens the Design Optimization tool and creates a new session. The model must be open or on the MATLAB path. <code>sdotool(sdosession)</code> opens a previously saved Design Optimization tool session.
Tips	<ul style="list-style-type: none">• <code>sdotool</code> also updates Signal Constraint blocks in the model to the equivalent blocks from the Signal Constraints block library.
Input Arguments	modelname Simulink model name, specified as a string inside single quotes (' '). sdosession Design Optimization tool session variable, saved in a MAT-file, model or MATLAB workspace.
Examples	Create a New Design Optimization Tool Session <pre>sdotool('pidtune_demo');</pre> Open Design Optimization Tool Using a Saved Session <pre>load sdoAircraft_sdosession; sdotool(SDOSessionData);</pre> <p><code>SDOSessionData</code> is the Design Optimization tool session variable saved in the <code>sdoAircraft_sdosession.mat</code> file.</p>
Tutorials	<ul style="list-style-type: none">• “Design Optimization to Meet a Custom Objective (GUI)”

sdouupdate

Purpose Update model containing Signal Constraint block

Syntax
`sdouupdate(modelname)`
`sdouupdate(modelname,noprompt)`
`session = sdouupdate(modelname)`

Description `sdouupdate(modelname)` replaces Signal Constraint blocks in a Simulink model with equivalent blocks from the **Signal Constraints** library. If the model has an associated response optimization project, this command replaces it with a session that you can use with the Design Optimization tool, after prompting you to update. The model must be open.

`sdouupdate(modelname,noprompt)` updates the response optimization project without prompting you.

`session = sdouupdate(modelname)` returns the Design Optimization tool session.

Input Arguments

modelname

Simulink model name that uses the parameter, specified as a string inside single quotes (' ').

noprompt

Whether to prompt you about updating the response optimization project (`false`) or not (`true`).

Default: `false`

Output Arguments

session

Design Optimization tool session name.

See Also

`sdotool`

Purpose Create Estimation Task in Control and Estimation Tools Manager GUI

Syntax `spetool('modelname')`

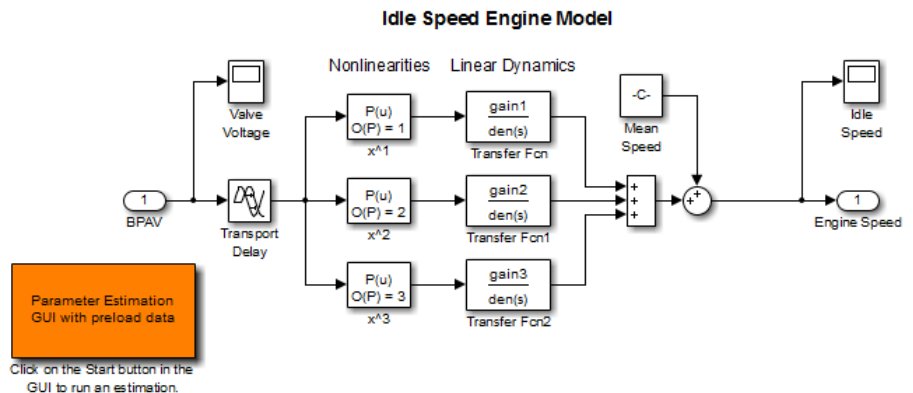
Description `spetool('modelname')` opens the Simulink model with the name `modelname` and creates an estimation task in the Control and Estimation Tools Manager GUI.

Examples Create an estimation task by typing the following command at the MATLAB prompt:

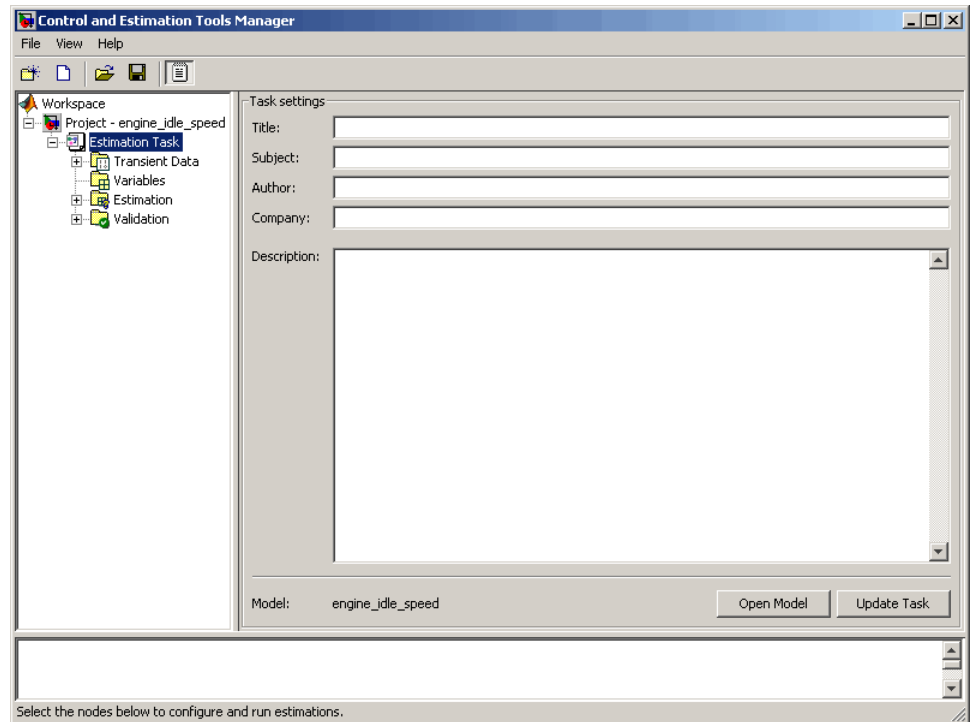
```
spetool('engine_idle_speed')
```

This command opens the following:

- Simulink model



- Control and Estimation Tools Manager containing a project with an estimation task



How To

- “Import Data (GUI)”