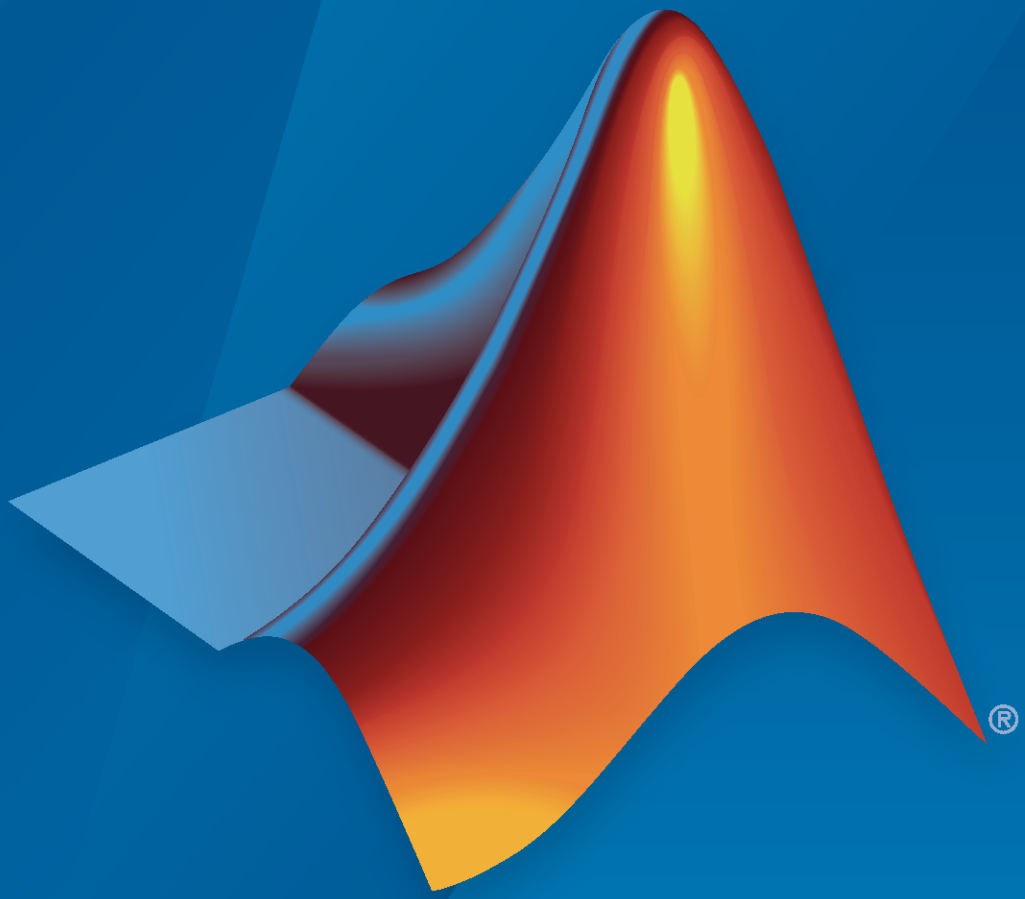
Simulink® Design Optimization™

Reference



MATLAB® & SIMULINK®

R2022a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Simulink® Design Optimization™ Reference

© COPYRIGHT 1998–2022 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	New for Version 2.0 (Release R2011b)
March 2012	Online only	Revised for Version 2.1 (Release R2012a)
September 2012	Online only	Revised for Version 2.2 (Release R2012b)
March 2013	Online only	Revised for Version 2.3 (Release R2013a)
September 2013	Online only	Revised for Version 2.4 (Release R2013b)
March 2014	Online only	Revised for Version 2.5 (Release 2014a)
October 2014	Online only	Revised for Version 2.6 (Release 2014b)
March 2015	Online only	Revised for Version 2.7 (Release 2015a)
September 2015	Online only	Revised for Version 2.8 (Release 2015b)
March 2016	Online only	Revised for Version 3.0 (Release 2016a)
September 2016	Online only	Revised for Version 3.1 (Release 2016b)
March 2017	Online only	Revised for Version 3.2 (Release 2017a)
September 2017	Online only	Revised for Version 3.3 (Release 2017b)
March 2018	Online only	Revised for Version 3.4 (Release 2018a)
September 2018	Online only	Revised for Version 3.5 (Release 2018b)
March 2019	Online only	Revised for Version 3.6 (Release 2019a)
September 2019	Online only	Revised for Version 3.7 (Release 2019b)
March 2020	Online only	Revised for Version 3.8 (Release 2020a)
September 2020	Online only	Revised for Version 3.9 (Release 2020b)
March 2021	Online only	Revised for Version 3.9.1 (Release 2021a)
September 2021	Online only	Revised for Version 3.10 (Release 2021b)
March 2022	Online only	Revised for Version 3.11 (Release 2022a)

Contents

1 Blocks

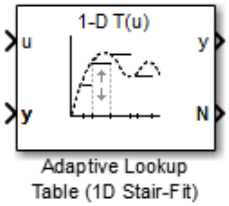
2 Classes

3 Methods and Apps

Blocks

Adaptive Lookup Table (1D Stair-Fit)

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

Parameters

First input (row) breakpoint set

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

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 `Tin`. 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

Select one of the following options:

- `Sample mean` — Average all the values received within a cell.
- `Sample mean (with forgetting)` — Give more weight to the new data. The weighting 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 `Tout` 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.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

See Also

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

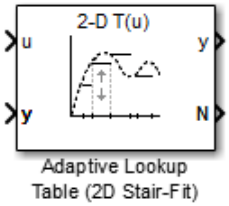
Topics

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

Introduced in R2009a

Adaptive Lookup Table (2D Stair-Fit)

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

Parameters

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.

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 `Tin`. 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 `Tout` 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.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

See Also

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

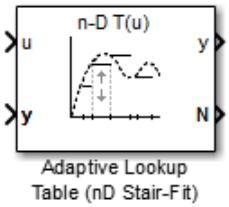
Topics

"What are Adaptive Lookup Tables?"
"Selecting an Adaptation Method"

Introduced in R2009a

Adaptive Lookup Table (nD Stair-Fit)

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

Parameters

Number of table dimensions

The number of dimensions for the adaptive lookup table.

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 `Tin`. 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 `Tout` for the adapted table.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

See Also

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

Topics

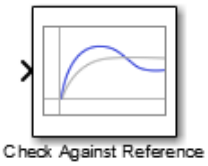
"Model Engine Using n-D Adaptive Lookup Table"

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

Introduced in R2009a

Check Against Reference

Check that model signal tracks reference signal during simulation
Library: Simulink Design Optimization / Model Verification
Simulink Design Optimization / Signal Constraints



Description

Check that a signal remains within the tolerance bounds of a reference signal during simulation.

- If the signal satisfies all bounds, the block does nothing.
- If the signal does not satisfy a bound, a warning message appears in the MATLAB® command window. You can also specify that the block:
  - Evaluate a MATLAB expression.
  - Stop the simulation and return an error message.

During simulation, the block can also return a logical assertion signal.

- If the signal satisfies all bounds, the assertion signal is true (`1`).
- If the signal does not satisfy a bound, the assertion signal is false (`0`).

You can add Check Against Reference blocks to multiple signals to check that they track reference signals. You can also plot the reference signal on a time plot to graphically verify the signal tracking.

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 the Bode magnitude, and optimize the model response to satisfy both time-domain 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.

Ports

Input

`Port_1 (u)` — Signal to check against specified reference signal
scalar | vector | matrix

Signal to check against the reference signal specified in **Amplitudes** and **Times** during simulation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point` | `enumerated` | `bus`

Output

`Port_1 (out)` — Output assertion signal
`1` | `0`

Output signal indicating if the input signal is within bounds of the specified assertion signal. At each time step, the output is returned as one of the following:

- True (`1`) if the assertion succeeds, that is, the input signal satisfies all bounds
- False (`0`) if the assertion fails, that is, the signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select Implement logic signals as Boolean data in Configuration Parameters.

Dependencies

To enable this port, on the **Assertion** tab, select the **Output assertion signal** parameter.

Data Types: `Boolean` | `double`

Parameters

`Show Plot` — Open plot window
`button`

Use the plot to view the following.

- System characteristics and signals computed during simulation — 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 on the **Bounds** tab or by right-clicking the plot and selecting **Bounds** > **New Bound**.

  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 . This action also linearizes the portion of the model between the specified linearization input and output.

- Adding a legend on the linear system characteristic plot by clicking .

`Show plot on block open` — Double-click block to open plot
`off` (default) | `on`

Open the plot window instead of the block parameters 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. If you want to access the block parameters from the plot window, select **Edit** or click .

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

Programmatic Use
Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer
`button`

Open the **Response Optimizer** to optimize the model response to meet design requirements specified on the **Bounds** tab.

See Also

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

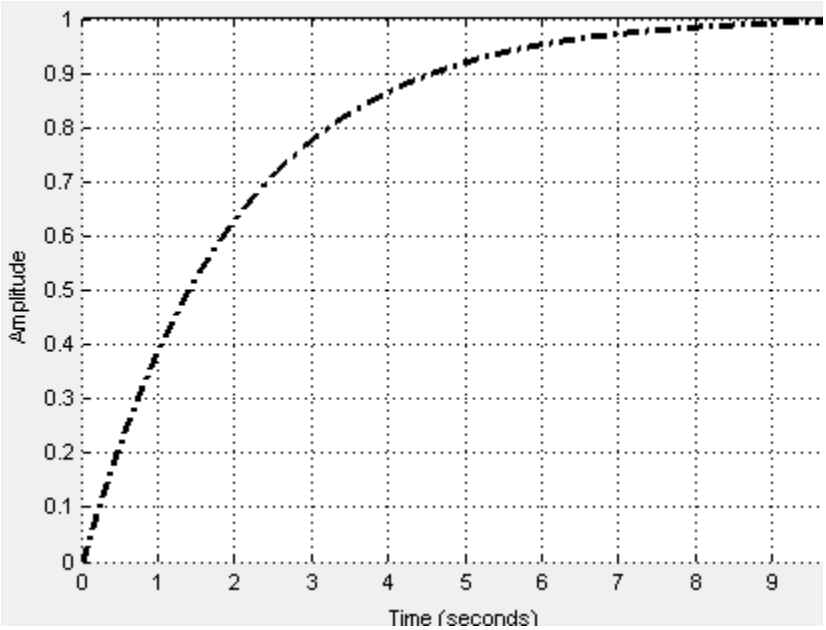
Bounds Tab

`Include reference signal tracking in assertion` — Check that signal tracks reference signal
`on` (default) | `off`

Check that the input signal to the block tracks the reference signal specified in **Amplitudes** and **Times** during simulation. The software displays a warning if the signal does not track the reference signal.

- When you disable this parameter, the software stops checking that the input signal tracks the reference during simulation.
- To view only the reference signal on the plot, disable **Enable assertion**.

The reference signal appears on a time plot if you select **Show plot**, as shown in the figure.



Dependencies

To use this parameter, select **Enable assertion** in block parameters. When **Enable assertion** is disabled, the software does not use the reference signal for assertion, but it continues to appear on the plot.

Programmatic Use

Parameter: `EnableReferenceBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Times` — Time vector

`linspace(0,10)` (default) | vector of positive values | MATLAB expression

Time vector for the reference signal in seconds, specified as a monotonically increasing vector of positive values. You can also specify a MATLAB expression that evaluates to a vector. Specify the corresponding amplitudes using the **Amplitudes** parameter. The **Times** vector must have the same dimension as the **Amplitudes** vector.

Programmatic Use

Parameter: `ReferenceTimes`
Type: vector
Value: vector of positive values | MATLAB expression
Default: `linspace(0,10)`

`Amplitudes` — Reference signal amplitude

`1-exp(-linspace(0,10)/2)` (default) | vector | MATLAB expression

Amplitude of the reference signal, specified as a real and finite vector, corresponding to the time vector specified in **Times**. You can also specify a MATLAB expression that evaluates to a vector. The **Amplitudes** vector must have the same dimension as the **Times (seconds)** vector.

Programmatic Use
Parameter: `ReferenceAmplitudes`
Type: vector
Value: vector | MATLAB expression
Default: `1-exp(-linspace(0,10)/2)`

`Absolute tolerance` — Absolute tolerance for signal

`eps^(1/3)` (default) | positive real scalar

Absolute tolerance used to determine bounds as the input signal approaches the reference signal, specified as a positive real scalar. During simulation, the signal must remain within upper and lower limits respective to the reference signal given by

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

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

Here:

- y_r is the value of the reference at a certain time.
- y_u is the upper tolerance bound corresponding to that time point.
- y_l is the lower tolerance bound corresponding to that time point.
- *RelTol* sets the relative tolerance between input and reference signal amplitudes at each time step. To set *RelTol*, use the **Relative tolerance** parameter.
- *AbsTol* sets the absolute tolerance between input and reference signal amplitudes at each time step. To set *AbsTol*, use the **Absolute tolerance** parameter.

The block asserts if the signal violates these limits.

Programmatic Use
Parameter: `AbsTolerance`
Type: scalar
Value: positive real scalar
Default: `eps^(1/3)`

`Relative tolerance` — Relative tolerance for signal

`0.01` (default) | positive real scalar

Relative tolerance used to determine bounds as the input signal approaches the reference signal, specified as a positive real scalar. During simulation, the signal must remain within upper and lower limits respective to the reference signal given by

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

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

Here:

- y_r is the value of the reference at a certain time.
- y_u is the upper tolerance bound.
- y_l is the lower tolerance bound.
- *RelTol* sets the relative tolerance between input and reference signal amplitudes at each time step. To set *RelTol*, use the **Relative tolerance** parameter.

- *AbsTol* sets the absolute tolerance between input and reference signal amplitudes at each time step. To set *AbsTol*, use the **Absolute tolerance** parameter.

The block asserts if the signal violates these limits.

Programmatic Use
Parameter: `RelTolerance`
Type: scalar
Value: positive real scalar
Default: `0.01`

Assertion Tab

`Enable assertion` — Enable or disable check

`on` (default) | `off`

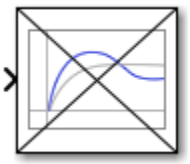
Enable the block to check that bounds specified and included for assertion on the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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 return an error message, by selecting **Stop simulation when assertion fails**.

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

When you disable this parameter, the block no longer checks that the input signal satisfies the specified bounds. The block icon also updates to indicate that assertion is disabled.



In the Simulink model, in the Configuration Parameters, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

Programmatic Use
Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`''` (default) | MATLAB expression

MATLAB expression to execute when assertion fails.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Halt simulation when assertion fails

`off` (default) | `on`

Stop the simulation when the input signal violates a bound specified on the **Bounds** tab during simulation, that is, when 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.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

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

- True (`1`) if assertion succeeds, that is, the input signal satisfies all bounds
- False (`0`) if assertion fails, that is, the input signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select Implement logic signals as Boolean data in Configuration Parameters.

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

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

Programmatic Use

Parameter: `export`

Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Code generation is available only when the **Output assertion signal** parameter is enabled.

See Also

Check Custom Bounds | Check Step Response Characteristics

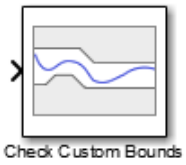
Topics

“Design Optimization to Track Reference Signal (GUI)”

Introduced in R2011b

Check Custom Bounds

Check that model signal satisfies bounds during simulation
Library: Simulink Design Optimization / Model Verification
Simulink Design Optimization / Signal Constraints



Description

Check that a signal satisfies the specified upper and lower bounds during simulation.

- If the signal satisfies all bounds, the block does nothing.
- If the signal does not satisfy a bound, a warning message appears in the MATLAB command window. You can also specify that the block:
  - Evaluate a MATLAB expression.
  - Stop the simulation and return an error message.

During simulation, the block can also return a logical assertion signal.

- If the signal satisfies all bounds, the assertion signal is true (`1`).
- If the signal does not satisfy a bound, the assertion signal is false (`0`).

Using the block, you can create:

- A single bound with one edge
- A single bound with multiple edges
- Multiple bounds

You can add Check Custom Bounds blocks to 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 the Bode magnitude, and optimize the model response to satisfy both time-domain 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.

Ports

Input

`Port_1 (u)` — Signal to check against specified bounds

scalar | vector | matrix

Signal to check against the upper and lower bounds specified in **Amplitudes** and **Times** during simulation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point` | `enumerated` | `bus`

Output

`Port_1 (out)` — Output assertion signal

`1` | `0`

Output signal indicating if the input signal is within bounds of the specified assertion signal. At each time step, the output is returned as one of the following:

- True (`1`) if the assertion succeeds, that is, the signal satisfies all bounds
- False (`0`) if the assertion fails, that is, the signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select **Implement logic signals as Boolean data** in Configuration Parameters.

Dependencies

To enable this port, on the **Assertion** tab, select the **Output assertion signal** parameter.

Data Types: `Boolean` | `double`

Parameters

`Show Plot` — Open plot window

`button`

Use the plot to view the following.

- System characteristics and signals computed during simulation — 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 on the **Bounds** tab or by right-clicking the plot and selecting **Bounds** > **New Bound**.

  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 . This action also linearizes the portion of the model between the specified linearization input and output.
- Adding a legend on the linear system characteristic plot by clicking .

`Show plot on block open` — Double-click block to open plot
`off` (default) | `on`

Open the plot window instead of the block parameters 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. If you want to access the block parameters from the plot window, select **Edit** or click .

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

Programmatic Use
Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer
`button`

Open the **Response Optimizer** to optimize the model response to meet design requirements specified on the **Bounds** tab.

See Also

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

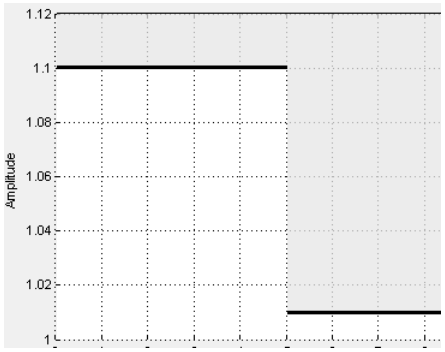
Bounds Tab

`Include upper bound in assertion` — Check that signal satisfies upper bound
`on` (default) | `off`

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

The bounds also appear on the time plot.

When you disable this parameter, the software stops checking that the input signal satisfies the upper bounds during simulation. The bound segments are also grayed out on the plot.



To view only the bounds and input signal on the plot, clear **Enable assertion**.

Dependencies

To use this parameter, on the **Assertion** tab, select **Enable assertion**. If you clear **Enable assertion**, the software does not use the bounds for assertion, but they continue to appear on the plot.

Programmatic Use

Parameter: `EnableUpperBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Times` — Upper bound time vector

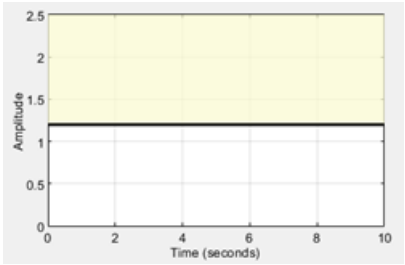
`[0 5; 5 10]` (default) | two-element vector | two-column matrix | cell array

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

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

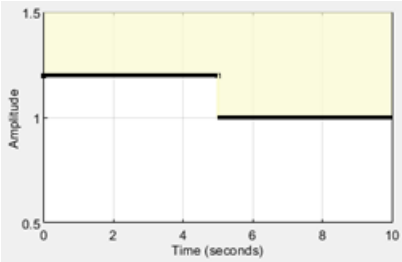
- To specify a single bound with one edge, use a two-element vector. The vector must be of the form $[T_{start}\ T_{end}]$, where T_{start} and T_{end} are positive finite numbers with $T_{start} < T_{end}$.

  For example, to create a single upper bound with an amplitude of 1.2 in a time interval of 0 to 10 seconds, specify `[0 10]` for **Times** and `[1.2 1.2]` for **Amplitudes**.



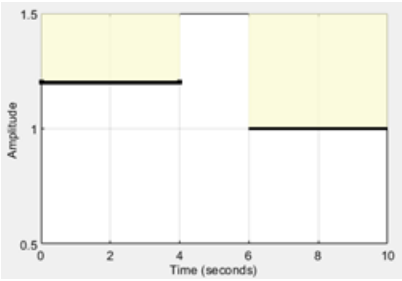
- To specify a single bound with multiple edges, use a two-column matrix. The matrix must be of the form $[T_1\ T_2\ ;\ T_2\ T_3\ ;\ldots;\ T_n\ T_{n+1}]$, where n is the number of edges in the bound. A single bound with multiple edges is piecewise continuous, so specify time intervals such that the next edge begins at the same time where the previous edge ends.

For example, to create a single bound with two edges at times of 0–5 and 5–10 seconds with amplitudes of 1.2 and 1, respectively, specify `[0 5;5 10]` for **Times** and `[1.2 1.2;1 1]` for **Amplitudes**.



- To specify multiple bounds, use a cell array of two-element vectors, two-column matrices, or both. That is, use a cell array of several single bounds.

  For example, to create two bounds at times of 0–4 and 6–10 seconds with amplitudes of 1.2 and 1, respectively, specify `{[0 4] [6 10]}` for **Times** and `{[1.2 1.2] [1 1]}` for **Amplitudes**.



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. 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.

  Click **Update Block** before simulating the model.

To check that the input signal satisfies the amplitudes that correspond to the upper bound time vectors, select both **Include upper bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `UpperBoundTimes`
Type: character vector
Value: two-element vector | two-column matrix | cell array
Default: `[0 5; 5 10]`

`Amplitudes` — Upper bound amplitude values

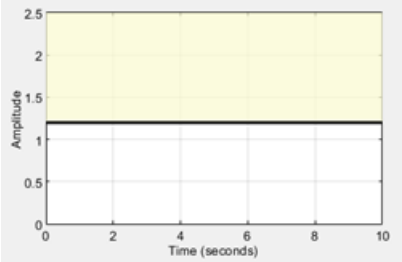
`[1.1 1.1; 1.01 1.01]` (default) | two-element vector | two-column matrix | cell array

Amplitude values for one or more upper bound segments.

Specify the corresponding start and end times in **Times**.

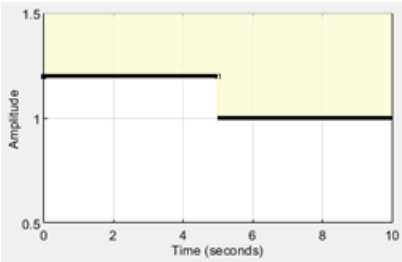
- To specify a single bound with one edge, use a two-element vector. The vector must be of the form $[A_1\ A_2]$, where A_1 and A_2 are the bound amplitudes at the corresponding edges of the time vector. For a constant amplitude throughout the time interval, use $A_1 = A_2$.

  For example, to create a single upper bound with an amplitude of 1.2 in a time interval 0 to 10 seconds, specify `[0 10]` for **Times** and `[1.2 1.2]` for **Amplitudes**.



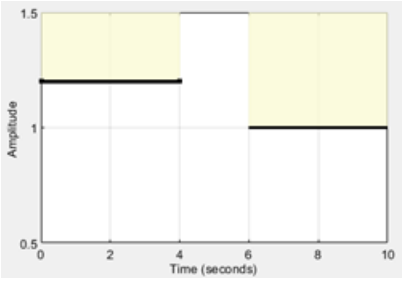
- To specify a single bound with multiple edges, use a two-column matrix. The matrix must be of the form $[A_1\ B_1\ ;\ldots;\ A_n\ B_n]$, where n is the number of edges in the bound. For a constant amplitude throughout a time interval, use $A_k = B_k$, where $k = 1,\ldots,n$

  For example, to create a single bound with two edges at times of 0–5 and 5–10 seconds with amplitudes of 1.2 and 1, respectively, specify `[0 5;5 10]` for **Times** and `[1.2 1.2;1 1]` for **Amplitudes**.



- To specify multiple bounds, use a cell array of two-element vectors, two-column matrices, or both. That is, use a cell array of several single bounds.

  For example, to create two bounds at times of 0–4 and 6–10 seconds with amplitudes of 1.2 and 1, respectively, specify `{[0 4] [6 10]}` for **Times** and `{[1.2 1.2] [1 1]}` for **Amplitudes**.



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. 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.

  Click **Update Block** before simulating the model.

To check that the input signal satisfies the upper amplitude bounds, select both **Include upper bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `UpperBoundAmplitudes`
Type: character vector
Value: two-element vector | two-column matrix | cell array
Default: `[1.1 1.1; 1.01 1.01]`

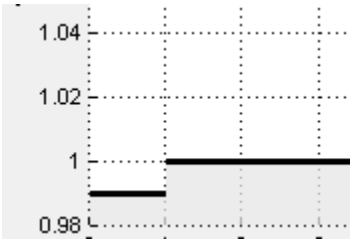
`Include lower bound in assertion` — Check that signal satisfies lower bound

`off` (default) | `on`

Check that a signal is greater than or equal to the lower bounds, specified in the lower bound **Times** and **Amplitudes**, during simulation. The software displays a warning if the signal violates the lower bounds.

The bounds also appear on the time plot.

When this parameter is disabled, the lower bounds are disabled and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.



To view only the bounds and input signal on the plot, clear **Enable assertion**.

Dependencies

To use this parameter, on the **Assertion** tab, select **Enable assertion**. If you clear **Enable assertion**, the software does not use the bounds for assertion, but they continue to appear on the plot.

Programmatic Use

Parameter: `EnableLowerBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Times` — Lower bound time vector

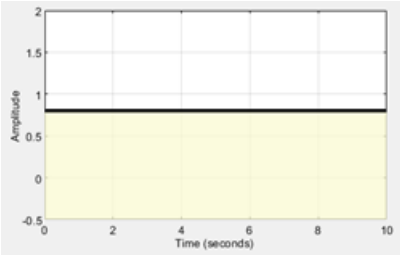
`[]` (default) | two-element vector | two-column matrix | cell array

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

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

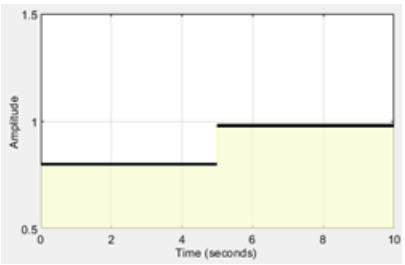
- To specify a single bound with one edge, use a two-element vector. The vector must be of the form [T_{start} T_{end}], where T_{start} and T_{end} are positive finite numbers with $T_{start} < T_{end}$.

  For example, to create a single lower bound with an amplitude of 0.8 in a time interval of 0 to 10 seconds, specify `[0 10]` for **Times** and `[0.8 0.8]` for **Amplitudes**.



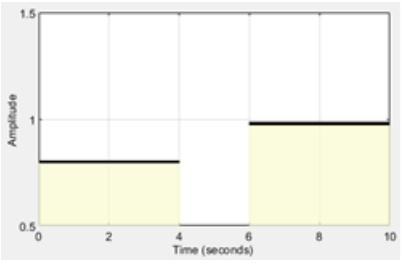
- To specify a single bound with multiple edges, use a two-column matrix. The matrix must be of the form [T_1 T_2 ; T_2 T_3 ;...; T_n T_{n+1}], where n is the number of edges in the bound. A single bound with multiple edges is piecewise continuous, specify time intervals such that the next edge begins at the same time where the previous edge ends.

  For example, to create a single bound with two edges at times of 0–5 and 5–10 seconds with amplitudes of 0.8 and 0.98, respectively, specify `[0 5;5 10]` for **Times** and `[0.8 0.8;0.98 0.98]` for **Amplitudes**.



- To specify multiple bounds, use a cell array of two-element vectors, two-column matrices, or both. That is, use a cell array of several single bounds.

  For example, to create two bounds at times of 0–4 and 6–10 seconds with amplitudes of 0.8 and 0.98, respectively, specify `{[0 4] [6 10]}` for **Times** and `{[0.8 0.8] [0.98 0.98]}` for **Amplitudes**.



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. 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.

  Click **Update Block** before simulating the model.

To check that the input signal satisfies the amplitudes that correspond to the lower bound time vectors, select both **Include lower bound in assertion** and **Enable assertion**.

Programmatic Use
Parameter: `LowerBoundTimes`
Type: character vector
Value: two-element vector | two-column matrix | cell array
Default: `[]`

`Amplitudes` — Lower bound amplitude values

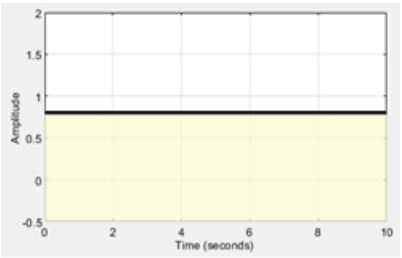
`[]` (default) | two-element vector | two-column matrix | cell array

Amplitude values for one or more lower bound segments.

Specify the corresponding start and end times in **Times**.

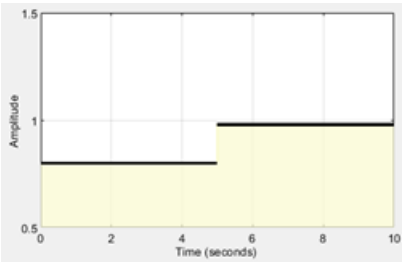
- To specify a single bound with one edge, use a two-element vector. The vector must be of the form $[A_1\ A_2]$, where A_1 and A_2 are the bound amplitudes at the corresponding edges of the time vector. For a constant amplitude throughout the time interval, use $A_1 = A_2$.

  For example, to create a single lower bound with an amplitude of 0.8 in a time interval of 0 to 10 seconds, specify `[0 10]` for **Times** and `[0.8 0.8]` for **Amplitudes**.



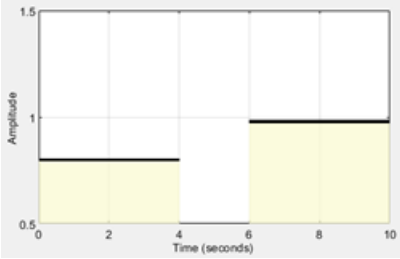
- To specify a single bound with multiple edges, use a two-column matrix. The matrix must be of the form $[A_1\ B_1\ ;\ldots;\ A_n\ B_n]$, where n is the number of edges in the bound. For a constant amplitude throughout a time interval, use $A_k = B_k$, where $k = 1,\ldots,n$

  For example, to create a single bound with two edges at times of 0–5 and 5–10 seconds with amplitudes of 0.8 and 0.98, respectively, specify `[0 5;5 10]` for **Times** and `[0.8 0.8;0.98 0.98]` for **Amplitudes**.



- To specify multiple bounds, use a cell array of two-element vectors, two-column matrices, or both. That is, use a cell array of several single bounds.

For example, to create two bounds at times of 0–4 and 6–10 seconds with amplitudes of 0.8 and 0.98, respectively, specify `{[0 4] [6 10]}` for **Times** and `{[0.8 0.8] [0.98 0.98]}` for **Amplitudes**.



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. 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.

  Click **Update Block** before simulating the model.

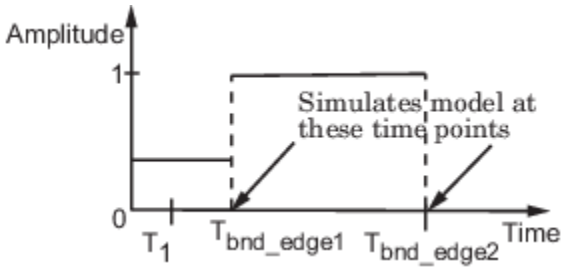
To check that the input signal satisfies the lower amplitude bounds, select both **Include lower bound in assertion** and **Enable assertion**.

Programmatic Use
Parameter: `LowerBoundAmplitudes`
Type: character vector
Value: two-element vector | two-column matrix | cell array
Default: `[]`

`Enable zero-crossing detection` — Evaluate model at bound edges

`on` (default) | `off`

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 an assertion that the signal satisfies that bound at its edge.



For more information, see “Zero-Crossing Detection”.

Programmatic Use
Parameter: `ZeroCross`
Type: character vector

Value: `'on'` | `'off'`
Default: `'on'`

Assertion Tab

`Enable assertion` — Enable or disable check

`on` (default) | `off`

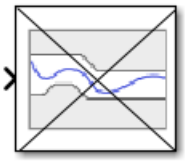
Enable the block to check that bounds specified and included for assertion on the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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 return an error message, by selecting **Stop simulation when assertion fails**.

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

When you disable this parameter, the block no longer checks that the input signal satisfies the specified bounds. The block icon also updates to indicate that assertion is disabled.



In the Simulink model, in the Configuration Parameters, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

Programmatic Use

Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`''` (default) | MATLAB expression

MATLAB expression to execute when assertion fails.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Halt simulation when assertion fails

`off` (default) | `on`

Stop the simulation when the signal violates a bound specified on the **Bounds** tab during simulation, that is, when 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.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

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

- True (`1`) if assertion succeeds, that is, the input signal satisfies all bounds
- False (`0`) if assertion fails, that is, the input signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select Implement logic signals as Boolean data in Configuration Parameters.

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

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

Programmatic Use

Parameter: `export`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Code generation is available only when the **Output assertion signal** parameter is enabled.

See Also

Check Against Reference | Check Step Response Characteristics

Introduced in R2011b

Check Step Response Characteristics

Check that model signal satisfies step response bounds during simulation
Library: Simulink Design Optimization / Model Verification
Simulink Design Optimization / Signal Constraints

Description

Check that a signal satisfies the step response bounds during simulation.

- If the signal satisfies all bounds, the block does nothing.
- If the signal does not satisfy a bound, a warning message appears in the MATLAB command window. You can also specify that the block:
  - Evaluate a MATLAB expression.
  - Stop the simulation and return an error message.

During simulation, the block can also return a logical assertion signal.

- If the signal satisfies all bounds, the assertion signal is true (`1`).
- If the signal does not satisfy a bound, the assertion signal is false (`0`).

You can add Check Step Response Characteristics blocks to 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 the Bode magnitude, and optimize the model response to satisfy both time-domain 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.

Ports

Input

`Port_1 (u)` — Signal to check against specified bounds

scalar | vector | matrix

Signal to check against the specified step response characteristics during simulation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point` | `enumerated` | `bus`

Output

`Port_1 (out)` — Output assertion signal

`1` | `0`

Output signal indicating if the input signal is within bounds of the specified assertion signal. At each time step, the output is returned as one of the following:

- True (`1`) if the assertion succeeds, that is, the signal satisfies all bounds
- False (`0`) if the assertion fails, that is, the signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select **Implement logic signals as Boolean data** in Configuration Parameters.

Dependencies

To enable this port, on the **Assertion** tab, select the **Output assertion signal** parameter.

Data Types: `Boolean` | `double`

Parameters

`Show Plot` — Open plot window

`button`

Use the plot to view the following.

- System characteristics and signals computed during simulation — 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 on the **Bounds** tab or by right-clicking the plot and selecting **Bounds > New Bound**.

  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 . This action also linearizes the portion of the model between the specified linearization input and output.
- Adding a legend on the linear system characteristic plot by clicking .

`Show plot on block open` — Double-click block to open plot
`off` (default) | `on`

Open the plot window instead of the block parameters 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. If you want to access the block parameters from the plot window, select **Edit** or click .

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

Programmatic Use
Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer
`button`

Open the **Response Optimizer** to optimize the model response to meet design requirements specified on the **Bounds** tab.

See Also

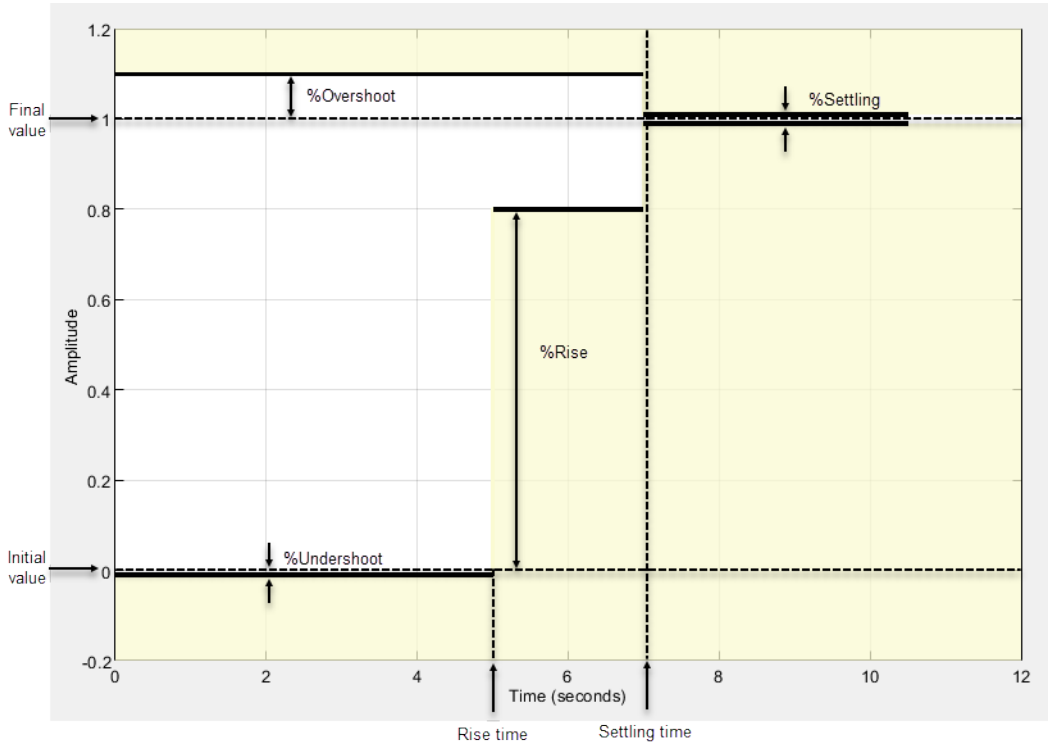
- “Design Optimization to Meet Step Response Requirements (GUI)”
- “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)”

Bounds Tab

`Include step response bound in assertion` — Check that signal satisfies step response bounds
`on` (default) | `off`

Check that the signal satisfies *all* the step response characteristics specified on the **Bounds** tab. The software displays a warning if the signal violates the specified step response characteristics.

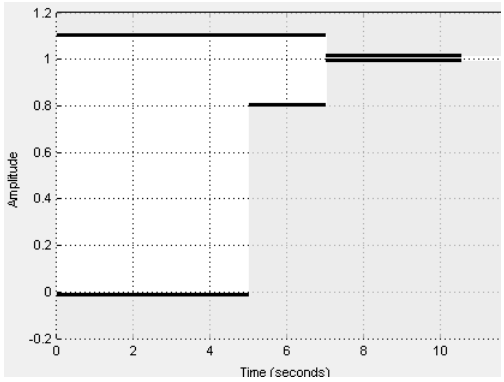
To view the specified step response bounds on a plot, click **Show Plot**.



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

- Amplitude greater than or equal to –0.01 up to the rise time of 5 seconds for 1% undershoot
- Amplitude greater than or equal to 0.8 between 5 and 7 seconds for 80% rise
- Amplitude less than or equal to 1.1 up to the settling time of 7 seconds for 10% overshoot
- Amplitude between 0.99 and 1.01 beyond the settling time for 1% settling

When you disable this parameter, the software stops checking that the input signal satisfies the bounds during simulation. The bound segments are also grayed out on the plot.



To view only the bounds and input signal on the plot, clear **Enable assertion**.

Dependencies

To use this parameter, on the **Assertion** tab, select **Enable assertion**. If you clear **Enable assertion**, the software does not use the bounds for assertion, but they continue to appear on the plot.

Programmatic Use

Parameter: `EnableStepResponseBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Step time` — Time when step response bounds start

`0` (default) | nonnegative scalar

Time, in seconds, when the step response bounds start, specified as a finite, real, nonnegative scalar.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the step time value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `StepTime`
Type: character vector
Value: nonnegative scalar
Default: `0`

`Initial value` — Signal value before step

`0` (default) | scalar

Value of the signal before the step occurs, specified as a finite real scalar not equal to the **Final value** parameter.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the initial value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `InitialValue`
Type: character vector
Value: scalar
Default: `0`

`Final value` — Signal value after step

`1` (default) | scalar

Value of the signal after the step occurs, specified as a finite real scalar not equal to the **Initial value** parameter.

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.

To check the input signal satisfies the final value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `FinalValue`
Type: character vector
Value: scalar
Default: 1

`Rise time` — Time taken to reach specified percentage of final value

5 (default) | positive scalar

Time taken, in seconds, for the signal to reach the percentage of the **Final value** specified in **% Rise**, specified as a finite, positive, real scalar less than the **Settling time** parameter.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the rise time value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `RiseTime`
Type: character vector
Value: positive scalar
Default: 5

`% Rise` — Percentage of final value for rise time

80 (default) | scalar in range 0–100

The percentage of the final value to use along with **Rise time** to define the overall rise time characteristics, specified as a real finite scalar in range 0–100 and less than 100 – **% Settling**.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the percent rise value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `PercentRise`
Type: character vector
Value: scalar in range 0–100
Default: 80

`Settling time (seconds)` — Time taken to reach settling range

7 (default) | positive scalar

The time taken, in seconds, for the signal to settle within a specified range around the final value, specified as a finite positive real scalar greater than **Rise time**. This settling range is defined as the

final value plus or minus the percentage of the final value. Specify the settling range in the **% Settling** parameter.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the settling time value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `SettlingTime`
Type: character vector
Value: positive scalar
Default: 7

`%` `Settling` — Settling range around final value

1 (default) | scalar in range 0–100

The percentage of the final value that defines the settling range of the **Settling time** characteristic, specified as a finite real scalar in the range 0–100. Specify a value less than both 100 – **% Rise** and **% Overshoot**.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the percent settling value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `PercentSettling`
Type: character vector
Value: scalar in range 0–100
Default: 1

`%` `Overshoot` — Amount by which signal can exceed final value

`10` (default) | scalar in range 0–100

The amount by which the signal can exceed the final value before settling, specified as a positive real scalar in the range 0–100 and greater than **% Settling**. This amount is specified as a percentage of step range. The step range is the difference between the final and initial values.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the percent overshoot value, select both **Include step response bound in assertion** and **Enable assertion**.

Programmatic Use

Parameter: `PercentOvershoot`
Type: character vector
Value: scalar in range 0–100
Default: `10`

`% Undershoot` — Amount by which signal can undershoot initial value

1 (default) | scalar in range 0–100

The amount by which the signal can undershoot the initial value before rising, specified as a positive real scalar in the range 0–100. This amount is specified as a percentage of the step range. The step range is the difference between the final and initial values.

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**. Click **Update Block** before simulating the model.

To check that the input signal satisfies the percent undershoot value, select both **Include step response bound in assertion** and **Enable assertion**.

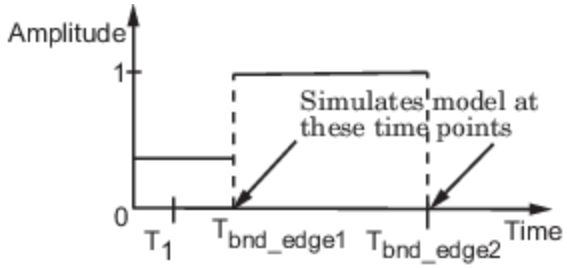
Programmatic Use

Parameter: `PercentUndershoot`
Type: character vector
Value: scalar in range 0–100
Default: 1

`Enable zero-crossing detection` — Evaluate model at bound edges

`on` (default) | `off`

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 an assertion that the signal satisfies that bound at its edge.



For more information, see “Zero-Crossing Detection”.

Programmatic Use

Parameter: `ZeroCross`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

Assertion Tab

`Enable assertion` — Enable or disable check

`on` (default) | `off`

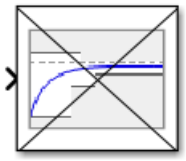
Enable the block to check that bounds specified and included for assertion on the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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 return an error message, by selecting **Stop simulation when assertion fails**.

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

When you disable this parameter, the block no longer checks that the input signal satisfies the specified bounds. The block icon also updates to indicate that assertion is disabled.



In the Simulink model, in the Configuration Parameters, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

Programmatic Use

Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails (optional)` — Expression to evaluate when assertion fails

`''` (default) | MATLAB expression

MATLAB expression to execute when assertion fails.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Halt simulation when assertion fails

`off` (default) | `on`

Stop the simulation when the input signal violates a bound specified on the **Bounds** tab during simulation, that is, when 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.

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

Dependencies

To enable this parameter, select **Enable assertion**.

Programmatic Use

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

`Output assertion signal` — Create output signal

`off` (default) | `on`

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

- True (`1`) if assertion succeeds, that is, the input signal satisfies all bounds
- False (`0`) if assertion fails, that is, the input signal violates a bound

The data type of the output signal is double by default. To set the output data type as Boolean for all blocks that generate logic signals in your Simulink model, select Implement logic signals as Boolean data in Configuration Parameters.

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

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

Programmatic Use

Parameter: `export`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Code generation is available only when the **Output assertion signal** parameter is enabled.

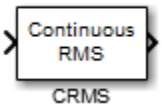
See Also

Check Against Reference | Check Custom Bounds

Introduced in R2011b

CRMS

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

Extended Capabilities

C/C++ Code Generation

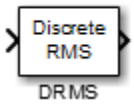
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

Introduced in R2009a

DRMS

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

Introduced in R2009a

Signal Constraint

Specify desired signal response

Compatibility

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

Library

Simulink Design Optimization

Introduced in R2009a

Classes

param.Continuous

Create continuous parameter

Description

A continuous parameter is a numeric parameter that can take any value in a specified interval. The parameter can be scalar-valued or matrix-valued.

Typically, you use continuous parameters to create parametric models and to estimate or optimize tunable parameters in such models.

Creation

Create a continuous parameter object in one of the following ways.

- `param.Continuous` function.
- `sdo.getParameterFromModel` to create objects for parameters in a Simulink model.

Syntax

```
p = param.Continuous(name)
p = param.Continuous(name,value)
```

Description

`p = param.Continuous(name)` creates a `param.Continuous` object and sets the `Name` property. The remaining properties of the object have default values.

`p = param.Continuous(name,value)` sets the `Name` and `Value` properties.

Properties

Free — Flag to make parameter tunable

`true` (`1`) (default) | `false` (`0`)

Flag to make the parameter tunable, specified as `true` (`1`) or `false` (`0`).

- Specify `true` (`1`) to make the parameter tunable.
- Specify `false` (`0`) to make the parameter nontunable.

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`.

`Info` — Parameter units and labels

structure

Parameter units and labels, specified as a structure array with fields `Label` and `Unit`.

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'`.

The default value for both the `Label` and `Unit` fields is `''`.

`Maximum` — Upper bound for parameter value

`Inf` (default) | scalar | matrix

Upper bound for the parameter value, specified as a scalar or matrix.

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`.

`Minimum` — Lower bound for parameter value

`-Inf` (default) | scalar | matrix

Lower bound for the parameter value, specified as a scalar or matrix.

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`.

`Name` — Parameter name

`''` (default) | character vector | string

This property is read-only.

Set this property at object creation.

`Scale` — Scaling factor

1 (default) | scalar | matrix

Scaling factor for normalizing the parameter value, specified as a scalar or matrix.

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`.

`Value` — Parameter value
`0` (default) | scalar | matrix

Parameter value, specified as a scalar or matrix.

Set the dimension of this property at object creation.

Object Functions

isreal Determine if parameter value, minimum, and maximum are real

Examples

Construct Continuous Parameter

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

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

See Also

`sdo.optimize` | `sdo.getParameterFromModel`

Topics
“Design Optimization to Meet Step Response Requirements (GUI)”
“Design Optimization to Meet Step Response Requirements (Code)”
“Design Optimization to Meet a Custom Objective (Code)”
Class Attributes
Property Attributes

Introduced in R2012b

param.State class

Package: `param`
Superclasses: `param.Continuous`

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:

- 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.

Examples

Get State Parameters from Model

```
modelname = 'sdoAircraft';
load_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`

Topics

“Estimate Model Parameter Values (Code)”
“Estimate Model Parameters and Initial States (Code)”
Class Attributes
Property Attributes

sdo.AnalyzeOptions class

Package: `sdo`

Analysis options for `sdo.analyze`

Syntax

```
opt = sdo.AnalyzeOptions
opt = sdo.AnalyzeOptions('Method',method_name)
```

Description

Specify analysis method and method options for sensitivity analysis using `sdo.analyze`.

Construction

`opt = sdo.AnalyzeOptions` creates an `sdo.AnalyzeOptions` object and assigns default values to the properties.

To change a property value, use dot notation. For example:

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

`opt = sdo.AnalyzeOptions('Method',method_name)` sets the value of the `Method` property to `method_name`.

Input Arguments

`method_name`

Method name, specified as one of the following values: `'Correlation'`, `'PartialCorrelation'`, `'StandardizedRegression'`, or `'All'`,

For example, `method_name = 'PartialCorrelation'`.

To use multiple methods, specify `method_name` as a cell array.

For information about each method, see the `Method` property description.

Properties

`Method`

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

- `'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:

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

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 and Machine Learning 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 and Machine Learning 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,...,xNp) and the corresponding single output, y. b_x is the regression coefficient vector calculated using least squares assuming a linear model $\widehat{y} = b_0 + \sum_{i=1}^{Np} \widehat{b}_{x_i} x_i$. 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`

Analysis method option that `sdo.analyze` uses, specified as one of the following values:

- `'Linear'` — Pearson analysis.

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 "Analyze Relation Between Parameters and Design Requirements".

Default: `'Linear'`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Specify Analysis Options

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

See Also

`sdo.analyze`

Topics

Class Attributes
Property Attributes
"Analyze Relation Between Parameters and Design Requirements"

sdo.EvaluateOptions class

Package: sdo

Cost function evaluation options for `sdo.evaluate`

Syntax

```
opt = sdo.EvaluateOptions
opt = sdo.EvaluateOptions(Name,Value)
```

Description

Specify options such as evaluation error handling, display settings, and the use of parallel computing for cost function evaluations using `sdo.evaluate`.

Construction

`opt = sdo.EvaluateOptions` creates an `sdo.EvaluateOptions` object and assigns default values to the properties.

`opt = sdo.EvaluateOptions(Name,Value)` creates an `sdo.EvaluateOptions` object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-12 and `Value` is the corresponding value.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-12 of `sdo.EvaluateOptions` object during object creation. For example, `opt = sdo.EvaluateOptions('Display','off')` creates a `sdo.EvaluateOptions` object specifying the `Display` property as `off`.

Properties

`UseParallel` — Parallel computing option

`false` or `0` (default) | `true` or `1`

Parallel computing option for `sdo.evaluate`, specified as one of the following:

- `false` or `0` — Do not use parallel computing during cost function evaluation.
- `true` or `1` — Use parallel computing during cost function evaluation.

  It is recommended that you also specify values for the `EvaluatedModel`, and `ParallelFileDependencies`, or `ParallelPathDependencies` properties, if needed.

Parallel Computing Toolbox™ software must be installed to enable parallel computing for the cost function evaluation.

`StopOnEvaluateError` — Handling of cost function evaluation error
`'off'` (default) | `'on'`

Handling of cost function evaluation error, specified as one of the following values:

- `'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.

`Display` — Viewing of display messages for cost function evaluations
`'final'` (default) | `'off'` | `'iter'`

Viewing of display messages for cost function evaluations, specified as one of the following values:

- `'final'` — Display only the final output.
- `'off'` — Display no output.
- `'iter'` — Display the output for each evaluation.

`ParallelFileDependencies` — File dependencies to use during parallel evaluation
`{}` (default) | cell array of character vectors

File dependencies to use during parallel evaluation, specified as a cell array of character vectors. Each character vector can specify either an absolute or relative path to a file. For example, `{'C:\matlab\work\file1.m','C:\matlab\myProject\file2.m'}`. These files are copied to the workers during parallel evaluation. Use `sdo.getModelDependencies` to find the dependencies of a Simulink model.

`ParallelPathDependencies` — Paths to dependencies to use during parallel evaluation
`{}` (default) | cell array of character vectors

Paths to dependencies to use during parallel evaluation, specified as a cell array of character vectors. For example, `{'C:\matlab\work','C:\matlab\myProject'}`. These path dependencies are temporarily added to the workers during parallel evaluation. Use `sdo.getModelDependencies` to find the dependencies of a Simulink model.

`EvaluatedModel` — Name of Simulink model to be evaluated
`''` (default) | character vector

Name of Simulink model to be evaluated, specified as a character vector. For example, `'sldo_model1'`.

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

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Create a Default Evaluation Options Object

Create an `sdo.EvaluateOptions` object.

```
opt = sdo.EvaluateOptions;
```

Specify handling of cost function evaluation error.

```
opt.StopOnEvaluateError = 'on';
```

Specify Cost Function Evaluation Options

Create an `sdo.EvaluateOptions` object to view output for each evaluation.

```
opt = sdo.EvaluateOptions('Display','iter');
```

See Also

`sdo.evaluate` | `sdo.getModelDependencies`

Topics

Class Attributes
Property Attributes
“Use Parallel Computing for Sensitivity Analysis”

sdo.Experiment

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

Description

Use an `sdo.Experiment` object to associate input and output data with signals in a Simulink model. Typically, you associate measured data that you collect from an experiment with the corresponding signals in the model.

You can use the `createSimulator` function of an experiment to create a simulation object. Use the simulation object to simulate the model and compare measured and simulated data. If the simulated response does not match the experimental data, you can estimate model parameter values for which the model response matches the measured data.

To specify model parameters for estimation, first create an estimation objective function, using the `sdo.Experiment` object and experiment design variables, to evaluate design requirements. You can then use `sdo.optimize` to estimate the parameter values that satisfy the design requirements. For more information about the workflow, see “Write a Cost Function” and “Estimate Model Parameter Values (Code)”.

Creation

Syntax

```
exp = sdo.Experiment(modelname)
```

Description

`exp = sdo.Experiment(modelname)` creates an `sdo.Experiment` object and sets the `ModelName` property. The remaining properties have default values.

Properties

`InitialStates` — Model initial state
`[]` (default) | `param.State` object | vector of `param.State` objects

Model initial state for the experiment, specified as a `param.State` object or vector of `param.State` objects.

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

To obtain model initial states from a Simulink model, 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 `getValuesToEstimate`. `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.

`InputData` — Experiment input data
`[]` (default) | data object

Experiment input data, specified as a data object. Use this property to specify signals to apply to root-level input ports. For information on supported forms of input data, see “Forms of Input Data”.

`ModelName` — Model name associated with experiment
`''` (default) | character vector | string

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

The model must be available on the MATLAB path.

Example: `'spe_engine_throttle'`

`OutputData` — Experiment output data
`[]` (default) | `Simulink.SimulationData.Signal` object | vector of `Simulink.SimulationData.Signal` objects

Experiment output data, specified as a `Simulink.SimulationData.Signal` object or vector of `Simulink.SimulationData.Signal` objects.

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

`Parameters` — Model parameter values
`[]` (default) | `param.Continuous` object | vector of `param.Continuous` objects

Model parameter values for the experiment, specified as a `param.Continuous` object or vector of `param.Continuous` objects.

- To specify a value for a single parameter, use a `param.Continuous` object.
- To specify values for multiple parameters, use a vector of `param.Continuous` objects.

To obtain model parameters from a Simulink model, use `sdo.getParameterFromModel`.

Use this property only for specifying parameter values that differ from the parameter 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 `getValuesToEstimate`. `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.

`Name` — Experiment name
`[]` (default) | character vector | string

Experiment name, specified as a character vector or string.

Example: `'Exp1'`

`Description` — Experiment description
`[]` (default) | character vector | string

Experiment description, specified as a character vector or string.

Example: `'Pendulum experiment 1'`

Object Functions

createSimulator	Create simulation object from experiment to compare measured and simulated data
getValuesToEstimate	Get model initial states and parameters for estimation from experiment
prepareToDeploy	Configure experiment for deployment with Simulink Compiler
setEstimatedValues	Update experiments with estimated model initial states and parameter values
updateIOData	Update experiment input and output data

Examples

Specify Input/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, respectively.

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  = 'outport';
VoltageSig.PortIndex = 1;
VoltageSig.Values    = timeseries(Charge_Data(:,3),Charge_Data(:,1));

exp.OutputData = VoltageSig;
```

Alternatives

“Estimate Parameters and States”

See Also

`sdo.optimize` | `sdo.getStateFromModel` | `param.Continuous` | `param.State`

Topics

“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)”
Class Attributes
Property Attributes

Introduced in R2012b

sdo.OperatingPointSetup class

Package: `sdo`

Set up steady-state operating point computation

Syntax

```
OpPointSetup = sdo.OperatingPointSetup(opSpec,inputsToUse,statesToUse)
OpPointSetup = sdo.OperatingPointSetup( ___ ,[findopOpts])
```

Description

An *operating point* of a dynamic system defines the states and root-level input signals of the model at a specific time. For example, in a car engine model, variables such as engine speed, throttle position, engine temperature, and surrounding atmospheric conditions typically describe the operating point.

A *steady-state operating point* of a model, also called an equilibrium or *trim* condition, includes state variables that do not change with time. A model can have several steady-state operating points. For example, a simple, damped pendulum has two steady-state operating points at which the pendulum position does not change with time.

Use `sdo.OperatingPointSetup` to set a model to steady-state as part of model optimization or evaluation. This step is useful when you are performing optimization or estimation using data measured when the system was at a non-zero steady state. Matching the model state to the system state, used for data collection, helps reduce model transients and improves results.

You need the Simulink Control Design toolbox to use `sdo.OperatingPointSetup`.

Construction

`OpPointSetup = sdo.OperatingPointSetup(opSpec,inputsToUse,statesToUse)` creates an `sdo.OperatingPointSetup` object using the operating point specifications `opSpec`, inputs `inputsToUse`, and states `statesToUse`.

`OpPointSetup = sdo.OperatingPointSetup( ___ ,[findopOpts])` creates an `sdo.OperatingPointSetup` object using additional arguments specified using `findopOpts`.

Input Arguments

opSpec — Operating point specifications

operating point specification object | [ ]

Operating point specifications, specified as an operating point specification object, or empty `[]`. Use `operspec` to create the `opSpec` object.

Also a property of the `sdo.operatingPointSetup` object. For more information, see OperatingPointSpec.

inputsToUse — Inputs to use for operating point setup

vector of indices | cell array of block paths | [ ]

Inputs to use for operating point setup, specified as a vector of indices, a cell array of block paths, or empty `[]`.

You can specify the inputs to use as:

- A vector of indices. For example, you can use the input argument `inputsToUse` as:

  ```
  inputsToUse = [2 3]
  ```

- A cell array of block paths. For example, you can use the input argument `inputsToUse` as:

  ```
  inputsToUse = {'modelname/in2','modelname/in3'}
  ```

- An empty array `[]` if you do not want to use any of the inputs.

When an input is supplied by experiment data in parameter estimation, that input should not be included among `inputsToUse`.

Also a property of the `sdo.operatingPointSetup` object. For more information, see UseOperatingPointInputs.

`statesToUse` — States to use for operating point setup
vector of indices | cell array of block paths | [ ]

States to use for operating point setup, specified as a vector of indices, a cell array of block paths, or empty `[]`.

You can specify the states to use as:

- A vector of indices. If you specify `statesToUse` as a vector of indices, the states should be in the same order as the states in `opSpec`. For example, you can use the input argument `statesToUse` as:

  ```
  statesToUse = [2 3]
  ```

- A cell array of block paths. For example, you can use the input argument `statesToUse` as:

  ```
  statesToUse = {'modelname/in2','modelname/in3'}
  ```

- An empty array `[]` if you do not use any of the states.

Also a property of the `sdo.OperatingPointSetup` object. For more information, see UseOperatingPointStates.

`findopOpts` — Trimming options to find operating point from specification
`findopOptions` option set

Trimming options to find operating point from specification, specified as a `findopOptions` option set. Use the `findopOptions` command to create the `findopOpts` option set.

Also a property of the `sdo.OperatingPointSetup` object. For more information, see FindopOptions.

Properties

`OperatingPointSpec` — Operating point specification for Simulink model
operating point specification object | [ ]

Operating point specification for Simulink model, specified as an operating point specification object, or empty `[]`.

Use `operspec` to create operating point specifications for your Simulink model, and create an operating point specification object, `opSpec`. For example, for components of the specification related to the states, you can change `Known`, `SteadyState`, `Min`, `Max`, `dxMin`, and `dxMax`.

You can modify the operating point specifications using dot notation. For example, if `opSpec` is the operating point specification object, `opSpec.States(1).x` accesses the state values of the first model state.

For more information on operating point specifications, see `operspec`.

`UseOperatingPointInputs` — Inputs to use for operating point setup
vector of indices | cell array of block paths | [ ]

Inputs to use for operating point setup, specified as a vector of indices, a cell array of block paths, or empty `[]`.

Use `UseOperatingPointInputs` to specify the inputs in the operating point to be applied to the model. When an input is supplied by experiment data, that input should not be included among `UseOperatingPointInputs`.

You can specify the inputs to use as:

- A vector of indices. For example, you can use the property `UseOperatingPointInputs` to specify the inputs to use as:

  ```
  OpPointSetup.UseOperatingPointInputs = [2 3]
  ```

  where `OpPointSetup` is the `sdo.OperatingPointSetup` object.
- A cell array of block paths. For example, you can use the property `UseOperatingPointInputs` to specify the inputs to use as:

  ```
  OpPointSetup.UseOperatingPointInputs = {'modelname/in2','modelname/in3'}
  ```

  where `OpPointSetup` is the `sdo.OperatingPointSetup` object.
- An empty array `[]` if you do not want to use any of the inputs.

`UseOperatingPointStates` — States to use for operating point setup
vector of indices | cell array of block paths | [ ]

States to use for operating point setup, specified as a vector of indices, a cell array of block paths, or empty `[]`.

Use `UseOperatingPointStates` to specify the states in the operating point to be applied to the model.

You can specify the states to use as:

- A vector of indices. If you specify `UseOperatingPointStates` as a vector of indices, the states should be in the same order as the states in `OperatingPointSpec`.For example, you can use the property `UseOperatingPointStates` to specify the states to use as:

  ```
  OpPointSetup.UseOperatingPointStates = [2 3]
  ```

where, `OpPointSetup` is the `sdo.OperatingPointSetup` object.

- A cell array of block paths. For example, you can use the property `UseOperatingPointStates` to specify the states to use as:

  ```
  OpPointSetup.UseOperatingPointStates = {'modelname/in2','modelname/in3'}
  ```

  where, `OpPointSetup` is the `sdo.OperatingPointSetup` object.
- An empty array `[]` if you do not use any of the states.

`FindopOptions` — Trimming options to find operating points from specifications
`findopOptions` option set

Trimming options to find operating point from specification, specified as a `findopOptions` option set.

Use the `findopOptions` command to create a `FindopOptions` option set for operating point computation. For more information, see `findopOptions`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create a Steady-State Operating Point Object

For this example, consider a Simulink model `'PopulationModel'` which models a simple ecology where an organism population growth is limited by the carrying capacity of the environment.

Set up your requirement or experiment, and then define a steady-state operating point object `OpPointSetup`. The operating point specification object is created using `operspec`. Use `sdo.OperatingPointSetup` to create the operating point object.

```
opSpec = operspec('PopulationModel');
inputsToUse = [];
statesToUse = 1;
OpPointSetup = sdo.OperatingPointSetup(opSpec,inputsToUse,statesToUse)
```

```
OpPointSetup =
  OperatingPointSetup with properties:

         OperatingPointSpec: [1x1 opcond.OperatingSpec]
    UseOperatingPointInputs: []
    UseOperatingPointStates: 1
              FindopOptions: []
```

Pass the operating point object `OpPointSetup` to your objective function, and include `OpPointSetup` in the call function to the `sim` method of the `sdo.SimulationTest` object. The `sim` method computes a steady-state operating point and applies it to the model. Specifically, it applies the operating point inputs specified in `UseOperatingPointInputs`, and applies the operating point states specified in `UseOperatingPointStates`. Then perform estimation or optimization per your requirement.

See Also

`findopOptions` | `operspec` | `sim` | `sdo.SimulationTest`

Topics

“Set Model to Steady-State When Estimating Parameters (Code)”
“What Is an Operating Point?” (Simulink Control Design)
“What Is a Steady-State Operating Point?” (Simulink Control Design)
“Compute Steady-State Operating Points” (Simulink Control Design)

Introduced in R2018a

sdo.OptimizeOptions

Optimization option set for `sdo.optimize` function

Description

Use an `sdo.OptimizeOptions` object to specify options for solving a design optimization problem using the `sdo.optimize` function. You can specify options such as a solver method, solver options, and the use of parallel computing during optimization.

Creation

Syntax

```
opt = sdo.OptimizeOptions
opt = sdo.OptimizeOptions(Name,Value)
```

Description

`opt = sdo.OptimizeOptions` creates a default option set for solving a design optimization problem using the `sdo.optimize` function. To modify the properties of this option set for your specific application, use dot notation.

`opt = sdo.OptimizeOptions(Name,Value)` creates an option set with properties on page 2-24 specified using one or more name-value arguments.

Properties

`Method` — Optimization solver

`'fmincon'` (default) | `'fminsearch'` | `'lsqnonlin'` | `'patternsearch'` | `'surrogateopt'`

Optimization solver that `sdo.optimize` uses to solve the optimization problem, specified as one of the following values:

- `'fmincon'` — Default for design optimization, derivative based method, allows nonlinear constraints and parameter bounds
- `'fminsearch'` — Derivative-free method
- `'lsqnonlin'` — Default for parameter estimation, works with gradient of residual between model output and data, allows parameter bounds
- `'patternsearch'` — Derivative-free method, allows nonlinear constraints and parameter bounds (requires Global Optimization Toolbox software)
- `'surrogateopt'` — Surrogate based method, allows nonlinear constraints, requires parameter bounds (requires Global Optimization Toolbox software)

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

`MethodOptions` — Optimization solver options
`[1x1 optim.options.Fmincon]` (default) | options object created using `optimoptions`

Optimization solver options, specified as an optimization options object, created using `optimoptions`. The options are configured based on the `Method` property. Because `'fmincon'` is the default optimization solver method, the default value for this property is an `optimoptions` object for `fmincon` solver. For information about the available optimization solver options, consult the options table for your solver based on the `Method` specified:

- `fmincon` `options`
- `fminsearch` `options`
- `lsqnonlin` `options`
- `patternsearch` `options`
- `surrogateopt` `options`

To modify the solver options, use dot notation. For example, `opt.MethodOptions.StepTolerance = 1.5e-3`.

`Restarts` — Number of times to restart optimization
`0` (default) | nonnegative integer

Number of times to restart optimization if convergence criteria are not satisfied, specified as a nonnegative integer. At each restart, the initial values of the tunable parameters are set to the final value of the previous optimization run.

`StopIfFeasible` — Option to stop optimization once feasible solution is found
`'on'` (default) | `'off'`

Option to stop optimization once a feasible solution satisfying constraints is found, specified as one of the following values:

- `'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.

`GradFcn` — Option to return gradient information from cost or constraint function
`'off'` (default) | `'on'`

Option to specify that the cost or constraint function you provide to `sdo.optimize` provides gradient information, specified as one of the following values:

- `'off'` — Your cost or constraint function does not provide gradient information. The software uses central differences to compute the gradients.
- `'on'` — Your cost or constraint function provides gradient information as one of its output arguments.

`UseParallel` — Parallel computing option
`false` or `0` (default) | `true` or `1`

Parallel computing option for `fmincon`, `lsqnonlin`, and `patternsearch` optimization solvers, specified as one of the following:

- `false` or `0` — Do not use parallel computing during optimization.
- `true` or `1` — Use parallel computing during optimization.

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

When set to `true`, the methods compute the following in parallel:

- `fmincon` — Finite difference gradients
- `lsqnonlin` — Finite difference gradients
- `patternsearch` — Poll and search set evaluation

Note Parallel computing is not supported for `fminsearch`.

For parallel computing, specify a simulation scenario for the Simulink model to optimize as an `sdo.SimulationTest` object in the `OptimizedModel` property. You can specify model file dependencies in the `ParallelFileDependencies` property, or you can specify paths to dependencies in the `ParallelPathDependencies` property, if needed.

`ParallelFileDependencies` — File dependencies to use during parallel optimization
`{}` (default) | cell array of character vectors

File dependencies to use during parallel optimization, specified as a cell array of character vectors. Each character vector can specify either an absolute or relative path to a file. These files are copied to the workers during parallel optimization. Use `sdo.getModelDependencies` to find the dependencies of a Simulink model.

Example: `opt.ParallelFileDependencies = {'C:\matlab\work\file1.m','C:\matlab\myProject\file2.m'}`

`ParallelPathDependencies` — Paths to dependencies to use during parallel optimization
`{}` (default) | cell array of character vectors

Paths to dependencies to use during parallel optimization, specified as a cell array of character vectors. If you do not want to copy the files to workers, use this property instead of the `ParallelFileDependencies` property. These path dependencies are temporarily added to the workers during parallel optimization. Use `sdo.getModelDependencies` to find the dependencies of a Simulink model.

Example: `opt.ParallelPathDependencies = {'C:\matlab\work','C:\matlab\myProject'}`

`OptimizedModel` — Name of Simulink model to optimize
`''` (default) | `sdo.SimulationTest` object | character vector

Name of Simulink model to optimize, specified as either an `sdo.SimulationTest` object or a character vector with the name of the model.

Specify `OptimizedModel` as an `sdo.SimulationTest` object when using both parallel optimization (`UseParallel = true`) and `fastRestart`.

Parallel Computing Toolbox software must be installed to enable parallel optimization.

Example: `Simulator = sdo.SimulationTest('model_demo')`

Object Functions

sdo.optimize Solve design optimization problem

Examples

Create Default Optimization Options Object

Create a default `sdo.OptimizeOptions` option set.

```
opt = sdo.OptimizeOptions;
```

Specify options using dot notation.

```
opt.Method = 'lsqnonlin';
opt.GradFcn = 'on';
```

Any property values you do not specify remain at their default values.

Specify Optimization Solver

Create an option set object that specifies nonlinear least-squares optimization solver as its solving method.

```
opt = sdo.OptimizeOptions('Method','lsqnonlin')
```

```
opt =
  OptimizeOptions with properties:

                    Method: 'lsqnonlin'
             MethodOptions: [1x1 optim.options.Lsqnonlin]
                  Restarts: 0
             StopIfFeasible: 'on'
                    GradFcn: 'off'
                UseParallel: 0
    ParallelPathDependencies: {}
    ParallelFileDependencies: {}
              OptimizedModel: ''
```

You can modify solver options using dot notation. For example, set the solver algorithm to the Levenberg-Marquardt method.

```
opt.MethodOptions.Algorithm = 'levenberg-marquardt';
```

View the method options.

```
opt.MethodOptions
```

```
ans =
  lsqnonlin options:
```

```
Options used by current Algorithm ('levenberg-marquardt'):
(Other available algorithms: 'trust-region-reflective')

Set properties:
                Algorithm: 'levenberg-marquardt'
                  Display: 'iter'
        FunctionTolerance: 1.0000e-03
   MaxFunctionEvaluations: Inf
            MaxIterations: 100
 SpecifyObjectiveGradient: 1
            StepTolerance: 1.0000e-03

Default properties:
           CheckGradients: 0
 FiniteDifferenceStepSize: 'sqrt(eps)'
     FiniteDifferenceType: 'forward'
                OutputFcn: []
                  PlotFcn: []
                 TypicalX: 'ones(numberOfVariables,1)'
              UseParallel: 0

Show options not used by current Algorithm ('levenberg-marquardt')
```

Any property values you do not specify remain at their default values.

See Also

`sdo.optimize` | `sdo.getModelDependencies` | `fastRestart`

Topics

“Specify Optimization Options”
“Speed Up Response Optimization Using Parallel Computing”
“Speed Up Parameter Estimation Using Parallel Computing”

Introduced in R2011a

sdo.ParameterSpace class

Package: sdo

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'})`.

`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 character vectors. For example, `{'Kp','Ki'}`.

This property is ready 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 `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'`, `'lhs'`, `'sobol'`, or `'halton'` 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 and to specify the degrees of freedom if using the t copula family.

Specify `[]` when the parameters are uncorrelated.

Default: `[]`

Options

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

Default: `sdo.SampleOptions`

Notes

Text notes associated with `ps`, specified as a character vector or cell array of character vectors. For example, `'notes for ps'`.

Default: ''

Default:

Methods

addParameter	Add parameter to `sdo.ParameterSpace` object
removeParameter	Remove parameter from `sdo.ParameterSpace` object
setDistribution	Set distribution of parameter in `sdo.ParameterSpace` object

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

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

`addParameter` | `sdo.sample` | `sdo.getParameterFromModel` | `makedist`

Topics
Class Attributes
Property Attributes

sdo.requirements.BodeMagnitude class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-32 of the requirement object during object creation. For example, `requirement = sdo.requirements.BodeMagnitude('Type','>=')` creates an `sdo.requirements.BodeMagnitude` object and specifies the `Type` property as a 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, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`FrequencyScale`

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points, specified as one of the following values:

- `'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`

Frequency units of the requirement, specified as one of the following values:

- `'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, specified as a character vector.

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

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.

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`

Topics

Class Attributes
Property Attributes

sdo.requirements.ClosedLoopPeakGain class

Package: `sdo.requirements`

Closed loop peak gain bound

Description

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 `sdo.optimize`.

You must have Simulink Control Design software to specify closed-loop peak gain bounds.

Construction

`pkgain_req = sdo.requirements.ClosedLoopPeakGain` creates a `sdo.requirements.ClosedLoopPeakGain` object and assigns default values to its properties.

`pkgain_req = sdo.requirements.ClosedLoopPeakGain(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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-36 of the requirement object during object creation. For example, `requirement = sdo.requirements.ClosedLoopPeakGain('Type','<=')` creates an `sdo.requirements.ClosedLoopPeakGain` object and specifies the `Type` property as an upper bound.

Properties

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

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, specified as a character vector.

Default: `''`

`PeakGain`

Peak gain bound.

Default: 2

`Type`

Peak gain requirement type, specified as one of the following:

- `'<='` — Upper bound
- `'=='` — 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.

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`

Topics

Class Attributes
Property Attributes

sdo.requirements.FunctionMatching class

Package: `sdo.requirements`

Impose function matching constraint on variable

Description

Use the `sdo.requirements.FunctionMatching` object to impose a function matching constraint on the values of a variable in a Simulink model. The variable can be a vector, matrix, or a multidimensional array that is a parameter in your model, such as the data of a lookup table in your model. You create the requirement object, and specify the linear or quadratic function that you want the variable to match. For example, for a two-dimensional variable, you can specify that test data from dependent variable `V` match a linear function of independent variables X_1 and X_2:

$$V = a_0 + a_1X_1 + a_2X_2$$

Where, a_0, a_1, and a_2 are the fit-coefficients, and X_1 and X_2 are vectors.

You use the `evalRequirement` method to evaluate whether your test data satisfies the specified requirement, and specify the independent variable vectors as inputs to the method. The software calculates the fit-coefficients using the independent variables and test data and then calculates the error between the test data and the specified function of the independent variables.

You can use the requirement object as an input to your cost function and use the `evalRequirement` command in the cost function to evaluate the requirement. You can then use the cost function and `sdo.optimize` to perform response optimization, subject to satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.FunctionMatching` creates an `sdo.requirements.FunctionMatching` requirement object and assigns default values to its properties. Use dot notation to customize the properties. Use the `evalRequirement` command to evaluate whether test data satisfies the specified requirement.

`requirement = sdo.requirements.FunctionMatching(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-40 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-40 of the requirement object during object creation. For example, `requirement = sdo.requirements.FunctionMatching('Type','quadratic')` creates an `sdo.requirements.FunctionMatching` object and specifies the function to be matched as quadratic.

Properties

`Centers` — Values to subtract from independent variable vectors

`[0 0]` (default) | vector

Values to subtract from the independent variable vectors that you input to the `evalRequirement` method, specified as a vector of length equal to number of independent variables. The number of independent variables equals the dimensionality of the test data. For example, suppose that you specify `Centers` as `[1 2]` for a two-dimensional variable with two independent variables. The software subtracts `1` from the first independent variable vector and `2` from the second independent variable vector.

Specify `Centers` to improve numerical conditioning when one or more independent variable vectors have a mean that differs from 0 by several orders of magnitude. If you do not specify independent variable vectors, then the software does not use `Centers`.

The default value of `Centers`, `[0 0]`, is for a two-dimensional variable. For variables of other dimensions, change the `Scales` and `Centers` properties together using the `set` command. For an example, see “Evaluate Function Matching Requirement for One-Dimensional Variable” on page 2-42.

Data Types: `double`

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`Method` — Method for processing errors

`'SSE'` (default) | `'SAE'` | `'Residuals'`

Method for processing errors during evaluation of requirement by `evalRequirement` command. The command computes an error signal that is the difference between test data and the function of the independent variables specified in the `Type` property. `Method` specifies how the errors are further processed. `Method` is specified as one of the following values:

- `'SSE'` — Sum of squares of the errors
- `'SAE'` — Sum of absolute values of errors
- `'Residuals'` — Errors

Data Types: `char`

`Name` — Name of requirement
`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

Data Types: `char`

`Scales` — Scaling of independent variable vectors
`[1 1]` (default) | vector of positive numbers

Scaling of the independent variable vectors that you input to the `evalRequirement` method, specified as a vector of length equal to number of independent variable. The number of independent variables equals the dimensionality of the test data. The independent variable vectors are divided by the corresponding `Scales` value after subtracting the `Centers` values.

For example, suppose that you specify `Centers` as `[5 50]` and `Scales` as `[10 100]` for a two-dimensional variable with two independent variables. The software subtracts `5` from the first independent variable vector and divides the result by `10`. The software subtracts `50` from the second independent variable vector and divides the result by `100`.

Specify `Scales` to improve numerical conditioning when independent variable vectors differ from each other by several orders of magnitude. If you do not specify independent variable vectors, then the software does not use `Scales`.

The default value of `Scales`, `[1 1]`, is for a two-dimensional variable. For variables of other dimensions, change the `Scales` and `Centers` properties together using the `set` command. For an example, see “Evaluate Function Matching Requirement for One-Dimensional Variable” on page 2-42.

Data Types: `double`

`Type` — Function to be matched
`'linear'` (default) | `'purequadratic'` | `'quadratic'`

Function to be matched, specified as one of the following:

- `'linear'` — Test data from dependent variable `V` are fit to a linear function. For example, for a two-dimensional variable with independent variables, X_1 and X_2, the linear function has the form:

  $$V = a_0 + a_1X_1 + a_2X_2$$

  When you use `evalRequirement` to evaluate the requirement for test data, the software calculates the fit coefficients a_0, a_1, and a_2 and then calculates the error between the test data and the linear function.
- `'purequadratic'` — Test data are fit to a quadratic function with no cross-terms. For a two-dimensional variable, the pure quadratic function has the form:

  $$V = a_0 + a_1X_1 + a_2X_1^2 + a_3X_2 + a_4X_2^2$$
- `'quadratic'` — Test data are fit to a quadratic function that includes cross-terms. For a two-dimensional variable, the quadratic function has the form:

  $$V = a_0 + a_1X_1 + a_2X_1^2 + a_3X_2 + a_4X_2^2 + a_5X_1X_2$$

If the test data are one-dimensional, there are no cross-terms and so the computation is the same as when `Type` is `'purequadratic'`.

Data Types: `char`

Methods

evalRequirement	Evaluate satisfaction of function matching requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create a Requirement Object to Match a Quadratic Function

Create a requirement object to impose a function matching requirement on the values of a variable.

```
Requirement = sdo.requirements.FunctionMatching;
```

The object is created with default properties and specifies that test data from the variable must match a linear function.

Specify that test data must match a quadratic function with no cross-terms.

```
Requirement.Type = 'purequadratic';
```

You can now use the `evalRequirement` command to evaluate whether test data satisfies the requirement.

Specify Scaling and Centering Values for Independent Variables

Create a function matching requirement object for a two-dimensional variable, and specify scaling and centering values for the independent variables.

The `Centers` and `Scales` properties are specified as vectors of length equal to number of independent variables. The number of independent variables equals the dimensionality of the test data.

```
Requirement = sdo.requirements.FunctionMatching('Centers',[5 10],...
    'Scales',[50 100]);
```

When you specify independent variables as inputs to the `evalRequirement` command, the software subtracts `5` from the first independent variable and then divides the result by `10`. The software subtracts `50` from the second independent variable and then divides the result by `100`.

Evaluate Function Matching Requirement for One-Dimensional Variable

Create a requirement object to match one-dimensional variable data to a linear function.

```
Requirement = sdo.requirements.FunctionMatching;
```

Specify the `Centers` and `Scales` properties for a one-dimensional variable by using the `set` command. You specify these properties because their default values are for a two-dimensional variable.

```
set(Requirement,'Centers',0,'Scales',1);
```

Specify test data for the one-dimensional variable.

```
dependentVariable = 0.5+5.*(1:5);
```

Evaluate the requirement.

```
evaluation = evalRequirement(Requirement,dependentVariable)
```

```
evaluation = 5.6798e-30
```

The software computes the linear function using the default independent variable vector `[0 1 2 3 4]` because you did not specify any independent variable vectors. There is one independent variable because the number of independent variables must equal the number of dimensions of the test data. The size of the independent variable vector equals the size of the test data.

In this example, the processing method has the default value of `'SSE'`, so `evaluation` is returned as a scalar value equal to the sum of squares of the errors. `evaluation` is very close to zero, indicating that the `dependentVariable` test data almost matches a linear function. Note that machine precision can affect the value of `evaluation` at such small values.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Write a Cost Function”

Introduced in R2016b

sdo.requirements.GainPhaseMargin class

Package: `sdo.requirements`

Gain and phase margin bounds

Description

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 `sdo.optimize`.

You must have Simulink Control Design software to specify gain and phase margin requirements.

Construction

`gainphase_req = sdo.requirements.GainPhaseMargin` creates a `sdo.requirements.GainPhaseMargin` object and assigns default values to its properties.

`gainphase_req = sdo.requirements.GainPhaseMargin(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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-44 of the requirement object during object creation. For example, `requirement = sdo.requirements.GainPhaseMargin('Type','>=')` creates an `sdo.requirements.GainPhaseMargin` object and specifies the `Type` property as a lower bound.

Properties

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

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`

`MagnitudeUnits`

Magnitude units of the requirement. Must be:

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

Default: `'db'`

`Name`

Requirement name, specified as a character vector.

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 specified as one of the following values:

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

Default: `'deg'`

`Type`

Gain and phase margin requirement type, specified as one of the following values:

- `'>='` — 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.

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`

Topics
Class Attributes
Property Attributes

sdo.requirements.MonotonicVariable class

Package: `sdo.requirements`

Impose monotonic constraint on variable

Description

Use the `sdo.requirements.MonotonicVariable` object to impose a monotonic constraint on a variable in your Simulink model. The variable can be a vector, matrix, or multidimensional array that is a parameter in your model, such as the breakpoints of a lookup table. You create the requirement object, and specify the type of monotonic requirement the variable needs to satisfy. For example, for a 2-D array variable, you can specify the elements of the first dimension as monotonically increasing and the elements of the second dimension as monotonically decreasing.

You can use the requirement object as an input to your cost function and use the `evalRequirement` command to evaluate whether your test data satisfies the specified requirement. You can then use the cost function and `sdo.optimize` to perform response optimization, subject to satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.MonotonicVariable` creates an `sdo.requirements.MonotonicVariable` requirement object and assigns default values to its properties. Use dot notation to customize the properties. Use the `evalRequirement` command to evaluate whether test data satisfies the specified requirement.

`requirement = sdo.requirements.MonotonicVariable(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-48 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-48 of the requirement object during object creation. For example, `requirement = sdo.requirements.MonotonicVariable('Type',{'>'})` creates an `sdo.requirements.MonotonicVariable` object and specifies the `Type` property as monotonically decreasing.

Properties

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`Name` — Name of requirement

`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

Data Types: `char`

`Type` — Monotonicity type

`{'<'}` (default) | cell array of character vectors

Monotonicity type for each dimension of the variable, specified as a cell array of character vectors. The size of the cell array equals the dimensions of the variable. Specify the required monotonic relation between the elements in each dimension as one of the following:

- `'<'` — Each element of the variable is less than the next element in that dimension. That is, the elements are monotonically increasing.
- `'<='` — Each element of the variable is less than or equal to the next element in that dimension.
- `'>'` — Each element of the variable is greater than the next element in that dimension. That is, the elements are monotonically decreasing.
- `'>='` — Each element of the variable is greater than or equal to the next element in that dimension.
- `'unconstrained'` — No constraint exists between the elements of the variable in that dimension. When the requirement is evaluated for test data using `evalRequirement`, the output corresponding to that dimension is `–Inf`, indicating the requirement is satisfied.

For example, for a two-dimensional variable, if you require the elements in the first dimension of the variable to be monotonically increasing while having no restriction on the elements of the second dimension, specify `Type` as `{'<','unconstrained'}`.

Data Types: `cell`

Methods

evalRequirement	Evaluate satisfaction of monotonic variable requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Monotonic Variable Requirement Object

Create a requirement object with default properties.

```
Requirement = sdo.requirements.MonotonicVariable;
```

Specify the requirement type for a 1-dimensional variable as monotonically decreasing.

```
Requirement.Type = {'>'};
```

You can now use the `evalRequirement` command to evaluate if test data satisfies the requirement.

Specify Monotonicity for Multidimensional Variable

Create a requirement object, and specify the monotonicity for a 3-dimensional variable.

```
Requirement = sdo.requirements.MonotonicVariable('Type',{'<','>','>='});
```

The object requires the elements in the first dimension of the variable to be monotonically increasing and the elements in the second dimension to be monotonically decreasing. Each element in the third dimension of the variable can be greater than or equal to the next element in that dimension.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Design Optimization Using Lookup Table Requirements for Gain Scheduling (Code)”
“Write a Cost Function”

Introduced in R2016b

sdo.requirements.PhasePlaneEllipse class

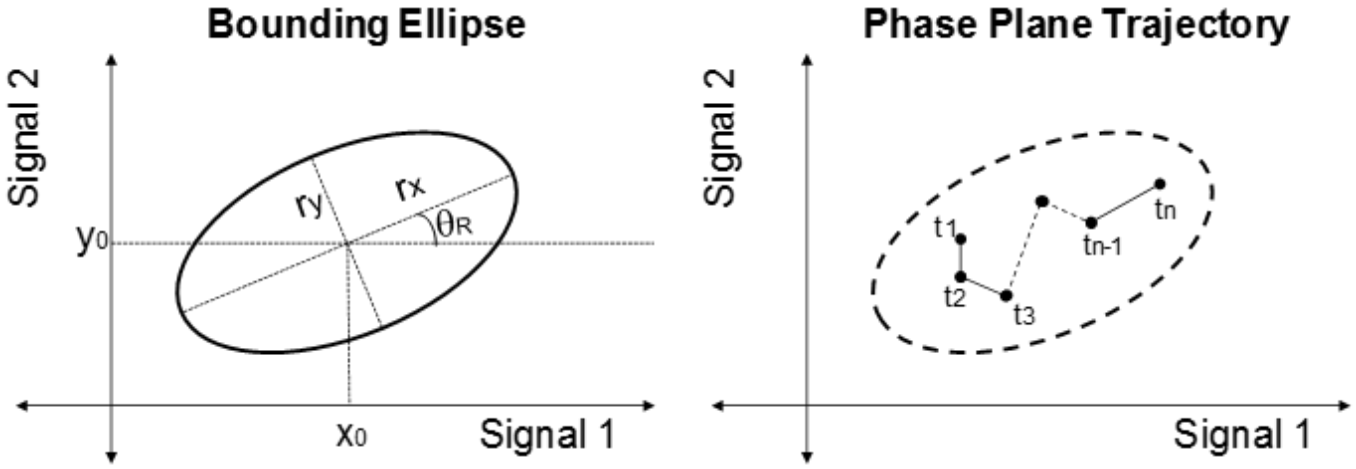
Package: `sdo.requirements`

Impose elliptic bound on phase plane trajectory of two signals

Description

Use the `sdo.requirements.PhasePlaneEllipse` object to impose an elliptic bound on the phase plane trajectory of two signals in a Simulink model. The phase plane trajectory is a plot of the two signals against each other. You specify the radii, center, and rotation of the bounding ellipse. You also specify whether you require the trajectory of the two signals to lie inside or outside the ellipse.

The following image shows the bounding ellipse and an example of the phase plane trajectory of two signals.



The X-Y plane is the phase plane defined by the two signals. r_x and r_y are the radii of the bounding ellipse along the x and y axes, and θ_R is the rotation of the ellipse about the center. The ellipse center is at (x_0,y_0). In the image, the phase plane trajectory of the signals lies within the bounding ellipse for all time-points t_1 to t_n.

You can use the object as an input to your cost function, and use the `evalRequirement` command in the cost function to evaluate whether your test signals satisfy the specified requirement. You can then use the cost function and `sdo.optimize` to perform parameter estimation or response optimization, subject to the satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.PhasePlaneEllipse` creates an `sdo.requirements.PhasePlaneEllipse` requirement object and assigns default values to its properties. Use dot notation to customize the properties.

Use the `evalRequirement` command to evaluate whether test signals satisfy the specified requirement.

`requirement = sdo.requirements.PhasePlaneEllipse(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-51 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-51 of the requirement object during object creation. For example, `requirement = sdo.requirements.PhasePlaneEllipse('Type','>=')` creates an `sdo.requirements.PhasePlaneEllipse` object and specifies the `Type` property as an outer bound.

Properties

`Center` — Location of center of ellipse

`[0 0]` (default) | 1-by-2 array

Location of center of bounding ellipse, specified as a 1-by-2 array with real finite values. The elements of the array specify x_0 and y_0, the x and y coordinates of the center location. To completely characterize the ellipse, also specify the `Radius` and `Rotation` properties of the ellipse. To see the equation of the ellipse, see “Description” on page 2-50.

Example: `[1.5,-1]`

Data Types: `double`

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`Method` — Method for requirement evaluation

`'Maximum'` (default) | `'Residuals'`

Method for requirement evaluation using the `evalRequirement` command, specified as one of the following:

- `'Maximum'` — The `evalRequirement` command computes the signed minimum distance of each point in the phase plane trajectory to the bounding ellipse and returns a scalar that is the maximum of these distances.

- `'Residuals'` — The `evalRequirement` command returns a column vector with the signed minimum distance of each point in the phase plane trajectory to the bounding ellipse. Use this method instead of `'Maximum'` to see the distance of all of the trajectory points to the phase plane ellipse.

Data Types: `char`

`Name` — Name of requirement
`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

Data Types: `char`

`Radius` — Radii of ellipse
`[1 0.5]` (default) | 1-by-2 array

Radii of ellipse, specified as a 1-by-2 array with real positive finite values. The elements of the array specify r_x and r_y, the x-axis and y-axis radii, before any rotation about the ellipse center. To completely characterize the ellipse, also specify the `Center` and `Rotation` properties of the ellipse. To see the equation of the ellipse, see “Description” on page 2-50.

Data Types: `double`

`Rotation` — Angle of rotation of ellipse about center
`0` (default) | real finite scalar

Angle of rotation θ_R of ellipse about center in radians, specified as a real finite scalar. The angle of rotation is specified from the x-axis. To completely characterize the ellipse, also specify the `Center` and `Radius` properties of the ellipse. To see the equation of the ellipse, see “Description” on page 2-50.

Example: `'Requirement 1 for myModel.'`

Data Types: `double`

`Type` — Type of bound
`'<='` (default) | `'>='`

Type of bound, specified as one of the following:

- `'<='` — Ellipse is an upper bound. The phase plane trajectory of the two signals should lie inside or on the ellipse.
- `'>='` — Ellipse is a lower bound. The phase plane trajectory of the two signals should lie outside or on the ellipse.

Methods

evalRequirement	Evaluate satisfaction of elliptical bound on phase plane trajectory of two signals

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Phase Plane Ellipse Requirement and Specify Ellipse Center

Create a requirement object with default properties.

```
Requirement = sdo.requirements.PhasePlaneEllipse;
```

The requirement object specifies the bounding ellipse as an upper bound with center located at [0,0], and no rotation. The x-axis radius of the ellipse is 1 and the y-axis radius is 0.5.

Specify the location of the center of the ellipse.

```
Requirement.Center = [1,0]
```

```
Requirement =
  PhasePlaneEllipse with properties:

         Radius: [1 0.5000]
         Center: [1 0]
       Rotation: 0
           Type: '<='
         Method: 'Maximum'
           Name: ''
    Description: ''
```

You can now use the `evalRequirement` command to evaluate whether test signals satisfy the requirement.

Specify Bounding Ellipse as Lower Bound

Create a requirement object, and specify the bounding ellipse as a lower bound. Use default values for the location of the center, radii, and rotation of the bounding ellipse.

```
Requirement = sdo.requirements.PhasePlaneEllipse('Type','>=');
```

The requirement object specifies that the phase plane trajectory of test signals should lie outside the ellipse.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Design Optimization Using Lookup Table Requirements for Gain Scheduling (Code)”
“Write a Cost Function”

Introduced in R2016b

sdo.requirements.PhasePlaneRegion class

Package: `sdo.requirements`

Impose region bound on phase plane trajectory of two signals

Description

Use the `sdo.requirements.PhasePlaneRegion` object to impose a region bound on the phase plane trajectory of two signals in a Simulink model. The phase plane trajectory is a plot of the two signals against each other. In the object, you can specify the bounded region as a single edge, or multiple piecewise-linear edges. You specify the starting and ending *x* and *y* coordinates of the bound edges, where the X-Y plane is the phase plane defined by the two signals. You also specify whether you require the trajectory of the two signals to lie inside or outside the bounded region specified by the edges.

You can use the object as an input to your cost function, and use the `evalRequirement` command in the cost function to evaluate whether your test signals satisfy the specified requirement. You can then use the cost function and `sdo.optimize` to perform parameter estimation or response optimization, subject to the satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.PhasePlaneRegion` creates an `sdo.requirements.PhasePlaneRegion` requirement object and assigns default values to its properties. Use dot notation to customize the properties of the object, except bound edges. To specify the bound edges simultaneously, use the `set` command. Use the `evalRequirement` command to evaluate whether test signals satisfy the specified requirement.

`requirement = sdo.requirements.PhasePlaneRegion(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-55 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-55 of the requirement object during object creation. For example, `requirement = sdo.requirements.PhasePlaneRegion('OpenEnd',[1 1])` creates an `sdo.requirements.PhasePlaneRegion` object and specifies that the first and last edge of the bound extend to infinity.

Properties

`BoundX` — X-coordinates of bound edges

`[-1 -1]` (default) | n-by-2 array

X-coordinates of edges that define the bounded region, specified as an *n*-by-2 array with finite values, where *n* is the number of edges in the bound. Each row of `BoundX` specifies the starting and ending x-coordinate values of an edge. The number of rows must match the number of rows in the `BoundY` property, which specifies the y-coordinates of the edges.

You must specify the `BoundX` and `BoundY` properties simultaneously, either using `Name,Value` arguments during object construction, or using the `set` command after object construction.

Data Types: `double`

`BoundY` — Y-coordinates of bound edges

`[0 1]` (default) | n-by-2 array

Y-coordinates of edges that define the bounded region, specified as an *n*-by-2 array with finite values, where *n* is the number of edges in the bound. Each row of `BoundY` specifies the starting and ending y-coordinate values of an edge. The number of rows must match the number of rows in the `BoundX` property, which specifies the x-coordinates of the edges.

You must specify the `BoundX` and `BoundY` properties simultaneously, either using `Name,Value` arguments during object construction, or using the `set` command after object construction.

Data Types: `double`

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`OpenEnd` — Extension of first and last bound edges to infinity

`[false false]` (default) | 1-by-2 logical array

Extension of first and last bound edges to infinity, specified as a 1-by-2 logical array.

If the first element of `OpenEnd` is `true`, the beginning of the first edge in the piecewise-linear bound extends to infinity. If the second element of `OpenEnd` is `true`, the end of the last edge in the piecewise-linear bound extends to infinity.

For an example, see “Create a Phase Plane Region Bound With Edge Extending to Infinity” on page 2-57.

Data Types: `logical`

`Name` — Name of requirement

`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

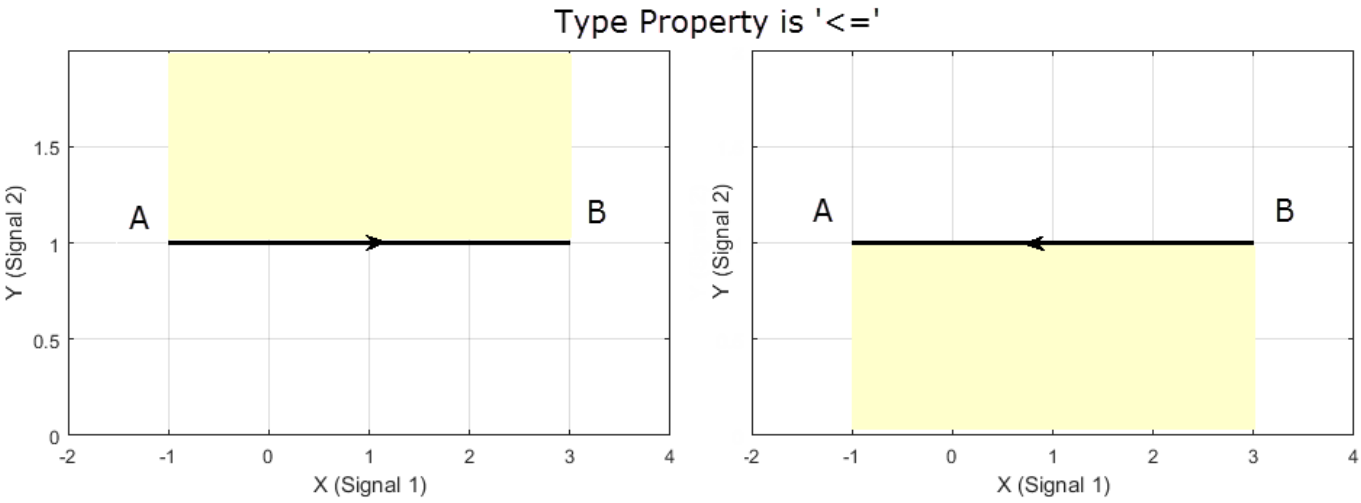
Data Types: `char`

Type — Type of bound
`'<='` (default) | `'>='`

Type of bound, specified as one of the following:

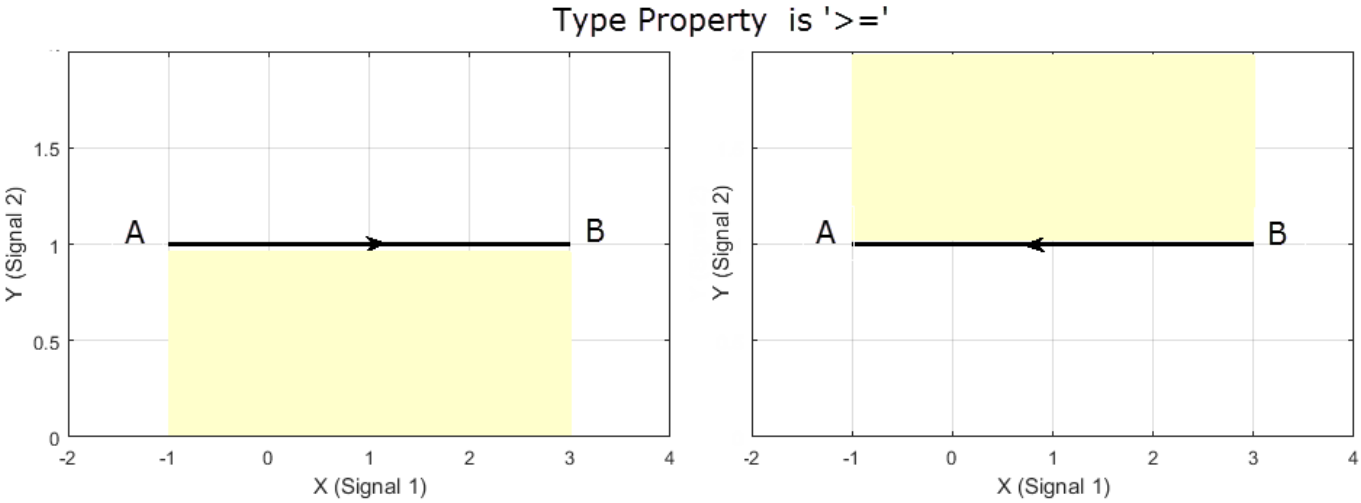
- `'<='` — The out-of-bound region is always to the left of each edge, where the forward direction is the direction of creation of the edge.

  For example, consider a single-edge bound between the point A located at (-1,1) and point B at (3,1). Suppose you specify the coordinates of point A before you specify point B. That is, you specify `BoundX` as `[-1 3]` and `BoundY` as `[1 1]`. The black arrow in the left plot shows the direction of creation of the edge, and the yellow region is the out-of-bound region when `Type` is `'<='`. Now suppose you instead specify point B before point A. That is, you specify `BoundX` is `[3 -1]` and `BoundY` is `[1 1]`. The right plot shows that the out-of-bound region is now below the edge because the direction of creation of the edge has reversed.



- `'>='` — The out-of-bound region is always to the right of each edge.

  For the single-edge example, suppose that you specify the coordinates of point A before you specify point B. In the left plot you can see that the yellow out-of-bound region is below the edge when `Type` is `'>='`. This is because the yellow region is to the right of the edge, in the direction of creation of the edge. If you instead specify point B before point A, the right plot shows that the out-of-bound region is now above the edge.



For an example with multiple bound edges, see “Specify Bounding Region With Multiple Edges and Evaluate the Requirement” on page 2-58.

Data Types: `char`

Methods

evalRequirement	Evaluate satisfaction of piecewise-linear bound on phase plane trajectory of two signals

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create a Phase Plane Region Bound With Edge Extending to Infinity

Create a phase plane region requirement object with default properties. The object specifies a piecewise-linear bound on the phase plane trajectory of two signals. The phase plane is the X-Y plane defined by the two signals.

```
Requirement = sdo.requirements.PhasePlaneRegion;
```

Specify a piecewise-linear bound with two edges. The (x,y) coordinates for the beginning and end of the first edge are (1,1) and (2,1). The second edge extends from (2,1) to (2,0). You must specify the `BoundX` and `BoundY` properties simultaneously.

```
set(Requirement,'BoundX',[1 2; 2 2],'BoundY',[1 1; 1 0])
```

Specify that the beginning of the first edge extends to infinity.

```
Requirement.OpenEnd = [1 0];
```

The first edge now extends from (-Inf,1) to (2,1).

You can now use the `evalRequirement` command to evaluate whether test data from two signals satisfy the requirement.

Specify Bounding Region With Multiple Edges and Evaluate the Requirement

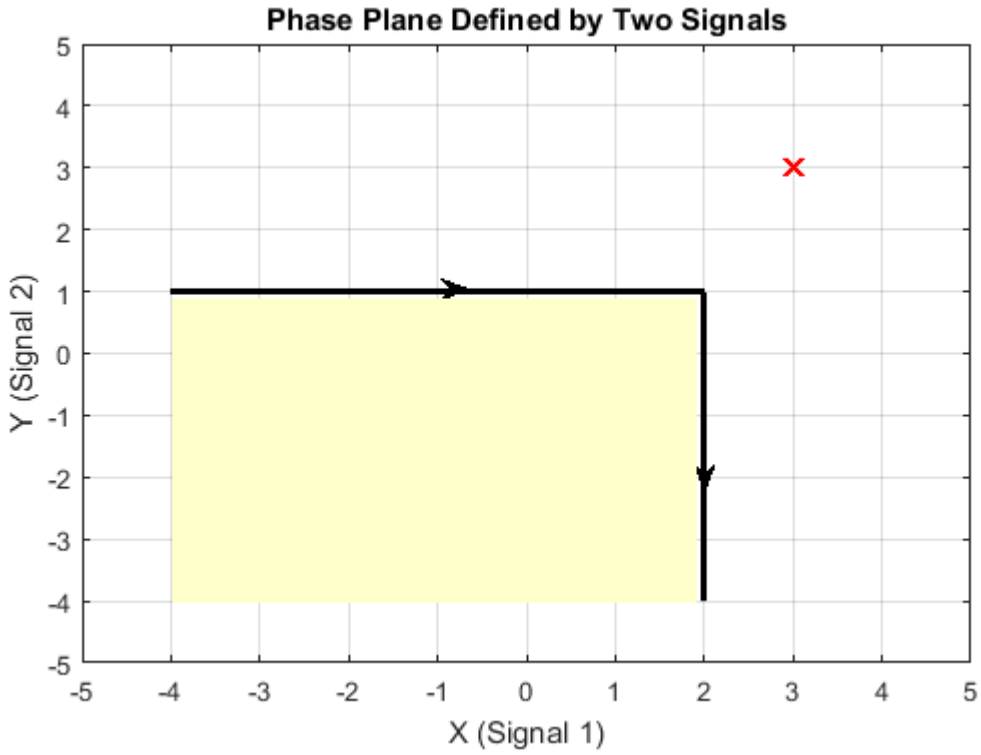
Create a requirement object to specify a piecewise-linear bound on the phase plane trajectory of two signals. The bound has two edges. The first edge extends from (-4,1) to (2,1). The second edge extends from (2,1) to (2,-4).

```
Requirement = sdo.requirements.PhasePlaneRegion('BoundX',[-4 2; 2 2],...
    'BoundY',[1 1; 1 -4]);
```

Specify the bound type as `'>='`.

```
Requirement.Type = '>=';
```

The plot below shows the bounding edges in black. The arrows indicate the direction in which the edges were specified. When you specify the `Type` property as `'>='`, the out-of-bound area is always to the right of each edge, where the forward direction is the direction of creation of the edge. As a result, the out-of bound region is the yellow shaded area, and a trajectory point located at (3,3) is in the bounded region.



Evaluate the requirement for the phase plane trajectory point located at (3,3).

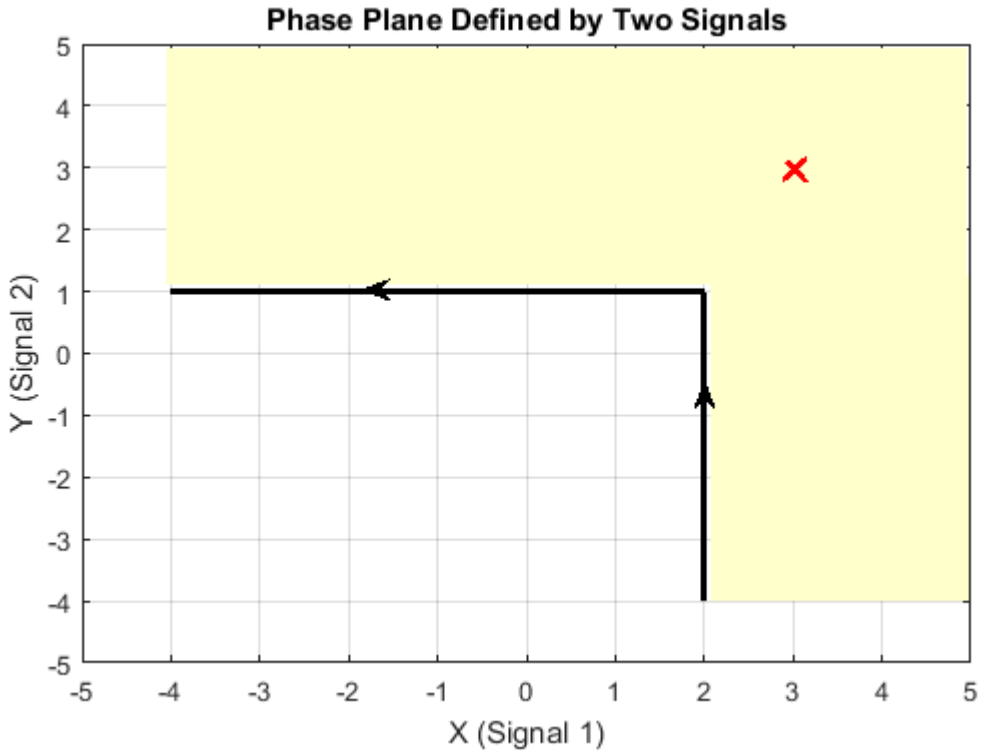
```
Evaluation = evalRequirement(Requirement,[3 3])
```

```
Evaluation = -0.6389
```

`evalRequirement` returns a negative number, indicating the requirement is satisfied.

Now create the requirement by changing the order of specification of the edges.

```
set(Requirement,'BoundX',[2 2; 2 -4],'BoundY',[-4 1;1 1]);
```



The plot shows that the edges were created in the opposite order. So, even though the requirement type is still `'>='`, the out-of-bound region, which is always to the right of the edges, is now flipped.

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,[3 3])
```

```
Evaluation = 0.1087
```

A positive `Evaluation` value indicates the requirement has been violated. Thus, for the same requirement type, the trajectory point at (3,3) is out of bounds when the edges are defined in the reverse order.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Write a Cost Function”

Introduced in R2016b

sdo.requirements.OpenLoopGainPhase class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-60 of the requirement object during object creation. For example, `requirement = sdo.requirements.OpenLoopGainPhase('Type','>=')` creates an `sdo.requirements.OpenLoopGainPhase` object and specifies the `Type` property as a lower bound.

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 `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 `nx2` 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, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`MagnitudeUnits`

Magnitude units of the requirement. Must be:

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

Default: `'db'`

`Name`

Requirement name, specified as a character vector.

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 specified as one of the following values:

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

Default: `'deg'`

Type

Gain and phase requirement type, specified as one of the following values:

- `'>='` — 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.

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`

Topics

Class Attributes
Property Attributes

sdo.requirements.PZDampingRatio class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-63 of the requirement object during object creation. For example, `requirement = sdo.requirements.PZDampingRatio('Type','>=')` creates an `sdo.requirements.PZDampingRatio` object and specifies the `Type` property as a lower bound.

Properties

`DampingRatio`

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

Default: `0.7071`

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

Name

Requirement name, specified as a character vector.

Default: `''`

Type

Damping ratio bound type, specified as one of the following values:

- `'<='` — 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.

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`

Topics
Class Attributes
Property Attributes

sdo.requirements.PZNaturalFrequency class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-65 of the requirement object during object creation. For example, `requirement = sdo.requirements.PZNaturalFrequency('Type','>=')` creates an `sdo.requirements.PZNaturalFrequency` object and specifies the `Type` property as a lower bound.

Properties

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`FrequencyUnits`

Frequency units of the requirement, specified as one of the following values:

- '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

Requirement name, specified as a character vector.

Default: ''

NaturalFrequency

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

Default: 2

Type

Natural frequency bound type, specified as one of the following values:

- '<=' — 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.

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`

Topics
Class Attributes
Property Attributes

sdo.requirements.PZSettlingTime class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-68 of the requirement object during object creation. For example, `requirement = sdo.requirements.PZSettlingTime('Type','>=')` creates an `sdo.requirements.PZSettlingTime` object and specifies the `Type` property as a lower bound.

Properties

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`Name`

Requirement name, specified as a character vector.

Default: `''`

`SettlingTime`

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

Default: 2

`TimeUnits`

Time units of the requirement, specified as one of the following values:

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

Default: `'second'`

`Type`

Settling time bound type, specified as one of the following values:

- `'<='` — 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.

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);
```

Alternatives

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

See Also

`copy` | `get` | `set`

Topics
Class Attributes
Property Attributes

sdo.requirements.RelationalConstraint class

Package: `sdo.requirements`

Impose relational constraint on pair of variables

Description

Use the `sdo.requirements.RelationalConstraint` object to impose a relational constraint on a pair of variables in a Simulink model. The variables can be any parameters in your model. You create the requirement object, and specify the type of relation you want between the elements of the two variables. For example, for two variables `var1` and `var2`, you can specify that each element of `var1` be greater than the corresponding element of `var2`.

You can use the requirement object as an input to your cost function and use the `evalRequirement` command to evaluate if your test data satisfies the specified requirement. You can then use the cost function and `sdo.optimize` to perform response optimization, subject to satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.RelationalConstraint` creates an `sdo.requirements.RelationalConstraint` requirement object and assigns default values to its properties. Use dot notation to customize the properties. Use the `evalRequirement` command to evaluate if test data satisfies the specified requirement.

`requirement = sdo.requirements.RelationalConstraint(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-72 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-72 of the requirement object during object creation. For example, `requirement = sdo.requirements.RelationalConstraint('Type','>')` creates an `sdo.requirements.RelationalConstraint` object and specifies that each data element in the first variable is strictly greater than the corresponding element in the second variable.

Properties

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`Name` — Name of requirement

`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

Data Types: `char`

`Type` — Relation type

`'<'` (default) | `'<='` | `'>'` | `'>='` | `'=='` | `'~='`

Relation type between the elements of the two variables, specified as one of the following:

- `'<'` — Each data element in the first variable is less than the corresponding element in the second variable.
- `'<='` — Each data element in the first variable is less than or equal to the corresponding element in the second variable.
- `'>'` — Each data element in the first variable is greater than the corresponding element in the second variable.
- `'>='` — Each data element in the first variable is greater than or equal to the corresponding element in the second variable.
- `'=='` — Each data element in the first variable is equal to the corresponding element in the second variable.
- `'~='` — Each data element in the first variable is not equal to the corresponding element in the second variable.

Data Types: `char`

Methods

evalRequirement	Evaluate satisfaction of relational constraint requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Constrain One Variable to be Greater Than Another

Create a requirement object with default properties to define the relation between two variables.

```
Requirement = sdo.requirements.RelationalConstraint;
```

Specify that the elements of the first variable be greater than the elements of the second variable.

```
Requirement.Type = '>';
```

You can now use the `evalRequirement` command to evaluate whether test data from two variables satisfy the requirement.

Constrain Two Variables to be Equal to Each Other

Create a requirement object, and specify the relation between two variables in your model.

```
Requirement = sdo.requirements.RelationalConstraint('Type','==');
```

The elements of the first variable are required to be equal to the corresponding elements of the second variable.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Write a Cost Function”

Introduced in R2016b

sdo.requirements.SignalBound class

Package: `sdo.requirements`

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 `–inf`.

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-74 of the requirement object during object creation. For example, `requirement = sdo.requirements.SignalBound('Type','>=')` creates an `sdo.requirements.SignalBound` object and specifies the `Type` property as a lower bound.

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 `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.

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, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`Name`

Requirement name, specified as a character vector.

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, specified as one of the following values:

- `'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, specified as one of the following values:

- `'<='` — 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`

Topics

Class Attributes
Property Attributes

sdo.requirements.SignalTracking class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-77 of the requirement object during object creation. For example, `requirement = sdo.requirements.SignalTracking('Type','>=')` creates an `sdo.requirements.SignalTracking` object and specifies the `Type` property as a lower bound.

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, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`InterpolationTimes`

Time points to use when comparing reference and testpoint signals, specified as one of the following values:

- `'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

Linear interpolation is used to compare the signals at the same timepoints.

Default: `'Reference only'`

`Method`

Algorithm for evaluating the requirement when the `Type` property is `'=='`, specified as one of the following values:

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

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.

Default: `'SSE'`

`Name`

Requirement name, specified as a character vector.

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'`	$F(x) = \sum_{t \in NOL} e(t) \times e(t)$ NOL is the set of nonoutlier samples.	$F(x) = \sum_{t \in OL} w \times \left\| e(t) \right\|$ w is a linear weight. OL is the set of outlier samples.
`'SAE'`	$F(x) = \sum_{t \in NOL} \left\| e(t) \right\|$ NOL is the set of nonoutlier samples.	$F(x) = \sum_{t \in OL} w$ w is a constant value. OL is the set of outlier samples.
`'Residuals'`	The software does not remove the outliers. $F(x) = \begin{bmatrix} e(0) \\ \vdots \\ e(N) \end{bmatrix}$ N is the number of samples.	

- `'off'`

Default: `'off'`

Type

Tracking requirement type, specified as one of the following values:

- `'=='` — 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.

Methods

evalRequirement	Evaluate tracking requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

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`

Topics
Class Attributes
Property Attributes

sdo.requirements.SingularValue class

Package: `sdo.requirements`

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-81 of the requirement object during object creation. For example, `requirement = sdo.requirements.SingularValue('Type','>=')` creates an `sdo.requirements.SingularValue` object and specifies the `Type` property as a 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, specified as a character vector. For example, `'Requirement on signal 1'`.

Default: `''`

`FrequencyScale`

Frequency-axis scaling.

Use this property to determine the value of the bound between edge start and end points, specified as one of the following values:

- `'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`

Frequency units of the requirement, specified as one of the following values:

- `'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, specified as a character vector.

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

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.

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`

Topics

Class Attributes
Property Attributes

sdo.requirements.SmoothnessConstraint class

Package: `sdo.requirements`

Impose bounds on gradient magnitude of variable

Description

Use the `sdo.requirements.SmoothnessConstraint` object to impose an upper bound on the gradient magnitude of a variable in a Simulink model. The variable can be a vector, matrix, or multidimensional array that is a parameter in your model, such as the data of a lookup table. For example, consider a car engine controller whose gain changes under different operating conditions determined by the car speed. You can use a gradient bound constraint to limit the rate at which the controller gain changes per unit change in vehicle speed.

You can use the requirement object as an input to your cost function and then use the `evalRequirement` command to evaluate whether your test data satisfies the requirement. If the test data is not smooth, the gradient of the test data is greater than the required bound. You can then use the cost function and `sdo.optimize` to perform response optimization, subject to satisfaction of the specified requirement. If you are performing sensitivity analysis, after you generate parameter samples, you can use the cost function and `sdo.evaluate` to evaluate the requirement for each generated sample.

Construction

`requirement = sdo.requirements.SmoothnessConstraint` creates an `sdo.requirements.SmoothnessConstraint` requirement object and assigns default values to its properties. Use dot notation to customize the properties. Use the `evalRequirement` command to evaluate whether test data satisfies the specified requirement.

`requirement = sdo.requirements.SmoothnessConstraint(Name,Value)` creates the requirement object with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-86 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-86 of the requirement object during object creation. For example, `requirement = sdo.requirements.SmoothnessConstraint('GradientBound',2.5)` creates an `sdo.requirements.SmoothnessConstraint` object and specifies the gradient magnitude bound as `2.5`.

Properties

`Description` — Requirement description

`''` (default) | character vector

Requirement description, specified as a character vector.

Example: `'Requirement 1 for myModel.'`

Data Types: `char`

`GradientBound` — Gradient magnitude bound

`1` (default) | nonnegative finite real scalar

Gradient magnitude bound, specified as a nonnegative finite real scalar. When you use the `evalRequirement` command to evaluate test data, the software checks whether the gradient magnitude of the test data is less than or equal to the specified bound. If the gradient of the test data is greater than the required bound, the test data is not smooth.

Data Types: `double`

`Name` — Name of requirement

`''` (default) | character vector

Name of requirement, specified as a character vector.

Example: `'Requirement1'`

Data Types: `char`

`Type` — Gradient magnitude bound type

`'<='` (default)

Gradient magnitude bound type, specified as lower bound, `'<='`. When you use the `evalRequirement` command, the software checks whether the gradient magnitude of the test data is less than or equal to the specified in `GradientBound` property.

Data Types: `char`

Methods

evalRequirement	Evaluate satisfaction of smoothness constraint requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Specify a Bound on the Gradient Magnitude of a Variable

Create a requirement object to impose a bound on the gradient magnitude of a variable. The object has default properties.

```
Requirement = sdo.requirements.SmoothnessConstraint;
```

Specify the gradient magnitude bound value.

```
Requirement.GradientBound = 5;
```

Alternatively, specify the bound during object creation.

```
Requirement = sdo.requirements.SmoothnessConstraint('GradientBound',5);
```

You can now use the `evalRequirement` command to evaluate whether test data satisfies the requirement.

See Also

`copy` | `get` | `set` | `evalRequirement`

Topics

“Write a Cost Function”

Introduced in R2016b

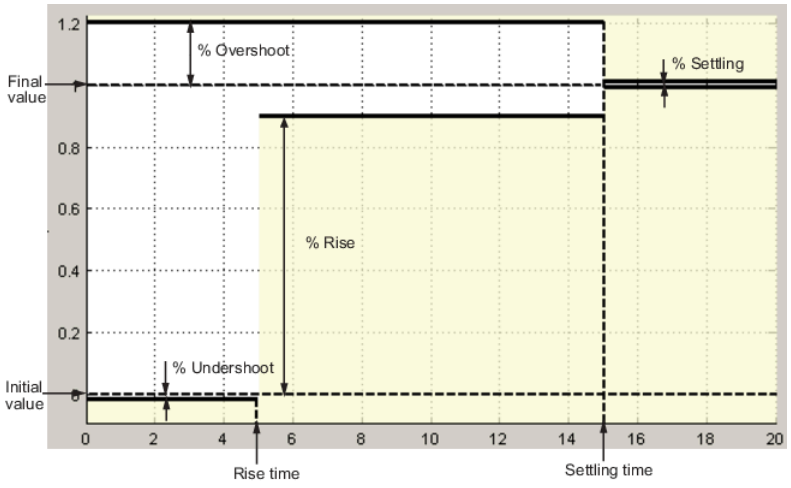
sdo.requirements.StepResponseEnvelope class

Package: `sdo.requirements`

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`.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-89 of the requirement object during object creation. For example, `requirement = sdo.requirements.StepResponseEnvelope('PercentOvershoot',20)` creates an `sdo.requirements.StepResponseEnvelope` object and specifies the `PercentOvershoot` property as 20.

Properties

`Description`

Requirement description, specified as a character vector. For example, `'Requirement on signal 1'`.

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, specified as a character vector.

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]`.

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, specified as one of the following values:

- `'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 satisfaction of step response requirement

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

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`

Topics

Class Attributes
Property Attributes

sdo.SampleOptions class

Package: `sdo`

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 values:

- `'random'` — Random samples are 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 are 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.
- `'sobol'` — Sobol quasirandom sequences are drawn from the probability distributions specified for the parameters. Use this option for highly systematic space-filling. Since Sobol method is deterministic, if you want slightly different sequences, modify the `MethodOptions` property. For more information, see “Generating Quasi-Random Numbers” (Statistics and Machine Learning Toolbox).

  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 Statistics and Machine Learning Toolbox software.
- `'halton'` — Halton quasirandom sequences are drawn from the probability distributions specified for the parameters. Like the Sobol method, you can use Halton method for highly

systematic space-filling. However, Sobol method gives more systematic space-filling if you have many parameters in your parameter set. Since Halton method is deterministic, if you want slightly different sequences, set the `MethodOptions` property. For more information, see “Generating Quasi-Random Numbers” (Statistics and Machine Learning Toolbox).

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 Statistics and Machine Learning Toolbox software.

- `'copula'` — Random samples are drawn from a copula. Use this option to impose correlations between the parameters using copulas. You specify the copula family and correlation type in the `MethodOptions` property. You must also specify the value of the `RankCorrelation` property of the `sdo.ParameterSpace` object that you use for sampling.

  Requires Statistics and Machine Learning Toolbox software.

For more information about the sampling methods, see “Generate Parameter Samples for Sensitivity Analysis”.

Default: `'random'`

`MethodOptions`

Sample method options, specified as a structure. `MethodOptions` is applicable only when `Method` is specified as `'sobol'`, `'halton'`, or `'copula'`.

- **`Method` is `'sobol'`** — Since Sobol method is deterministic, if you want to generate slightly different sequences, modify the default values in `MethodOptions`. Specify `MethodOptions` as a structure with the following fields:

  - `Skip` — Number of initial points to ignore in a Sobol sequence of points, specified as a non-negative integer. The default value is `1`.
  - `Leap` — Number of points to ignore between selected points in a Sobol sequence, specified as a non-negative integer. The default value is `0`.
  - `ScrambleMethod` — Shuffling of the Sobol sequence points, specified as a structure with following fields:

    - `Type` — Name of the scramble method, specified as `'MatousekAffineOwen'` (Matousek-Affine-Owen scrambling algorithm [1]). Sobol sets with scrambling are not deterministic. Successive runs using this algorithm generate different points. To always generate the same Sobol sequence points, reset the random number generator each time using the `rng` command.
    - `Options` — Specify as an empty cell array.

    For example, specify `ScrambleMethod` as `struct('Type','MatousekAffineOwen',Options',{{}})`.

    If you do not want to scramble the sequence, specify `ScrambleMethod` as `[]`.

    The default value for `ScrambleMethod` is `0x0 struct`.
  - `PointOrder` — Order in which the Sobol sequence points are produced, specified as one of the following:

- `'standard'` — Points produced match the original Sobol sequence implementation.
- `'graycode'` — Sobol sequence is generated using an implementation that uses the Gray code of the index instead of the index itself.

The default value for `PointOrder` is `'standard'`.

Sobol method is used for a highly systematic space-filling. However, some combinations of the `MethodOptions` values may result in sequence points that are clustered and not space-filling. After you have generated the samples using `sdo.sample`, view the generated samples to ensure that they are space-filling.

- **`Method` is `'halton'`** — Since Halton method is deterministic, if you want to generate slightly different sequences, modify the default values in `MethodOptions`. Specify `MethodOptions` as a structure with the following fields:

  - `Skip` — Number of initial points to ignore in a Halton sequence of points, specified as a non-negative integer. The default value is `1`.
  - `Leap` — Number of points to ignore between selected points in a Halton sequence, specified as a non-negative integer. The default value is `0`.
  - `ScrambleMethod` — Shuffling of the Halton sequence points, specified as a structure with following fields:

    - `Type` — Name of the scramble method, specified as `'RR2'` (reverse-radix algorithm [2]).
    - `Options` — Specify as an empty cell array.

    For example, specify `ScrambleMethod` as `struct('Type','RR2',Options',{{}})`.

    If you do not want to scramble the sequence, specify `ScrambleMethod` as `[]`.

    The default value for `ScrambleMethod` is `0x0 struct`.

  Halton method is used for a highly systematic space-filling. However, some combinations of the `MethodOptions` values may result in sequence points that are clustered and not space-filling. After you have generated the samples using `sdo.sample`, view the generated samples to ensure that they are space-filling.

- **`Method` is `'copula'`** — `MethodOptions` is a structure with the following fields:

  - `Family` — Copula family, specified as one of the following values:

    - `'Gaussian'` — Gaussian copula
    - `'t'` — t copula

    The default value is `'Gaussian'`.

  - `Type` — Rank correlation type, specified as one of the following values:

    - `'Spearman'` — Spearman’s rank correlation
    - `'Kendall'` — Kendall’s rank correlation

    The default value is `'Spearman'`.

  - `DOF` — Degrees of freedom of t copula, specified as a positive number.

    For a Gaussian copula, specify `DOF` as `[]`. Specification of `DOF` is required for a t copula.

The default value is `[]`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

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;
opt.Method = 'lhs';
```

Specify Sobol Sampling method

Create a default option set.

```
opt = sdo.SampleOptions;
```

Specify the sampling method as Sobol.

```
opt.Method = 'sobol';
```

Specify a scrambling method.

```
opt.MethodOptions.ScrambleMethod = struct('Type','MatousekAffineOwen','Options',{{}});
```

Specify Copula-Based Sampling Method

```
opt = sdo.SampleOptions;
opt.Method = 'copula';
opt.MethodOptions.Family = 't';
opt.MethodOptions.DOF = 2;
```

References

[1] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, Number 4, 1998, pp. 527–556.

[2] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, Number 2, 1997, pp. 266–294.

See Also

`sdo.sample`

Topics

“Generate Parameter Samples for Sensitivity Analysis”

sdo.SimulationTest class

Package: `sdo`

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. You can also specify linear systems to compute if you have Simulink Control Design toolbox. 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 `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 character vector or string. For example, `'sdoHydraulicCylinder'`.

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

`SystemLoggingInfo`

Linear system logging settings.

This property is a vector of `sdo.SystemLoggingInfo` objects.

If you specify the `SystemLoggingInfo` property, the `sim` method linearizes the model during simulation.

Note You can also use `linearize` command from Simulink Control Design to compute linear systems. However, to use fast restart, you must use `SystemLoggingInfo` property and `sim` instead.

Default: `[]`

`ModelName`

Simulink model name associated with the simulation scenario. The model must be on the MATLAB path.

`Name`

Name of the scenario

Default: `''`

`Parameters`

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

fastRestart	Simulate Simulink model in fast restart mode using simulation scenario
find	Find logged data set
prepareToDeploy	Configure simulation scenario for deployment with Simulink Compiler
sim	Simulate Simulink model using simulation scenario
who	List logged data names

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Create Simulation Scenario for Model

Create a simulation scenario for a model.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;
Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly';
Pressures.OutputPortIndex = 1;
simulator = sdo.SimulationTest('sdoHydraulicCylinder');
```

Specify model signals to log.

```
simulator.LoggingInfo.Signals = [Pressures];
```

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);
```

Alternatives

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

See Also

`sdo.optimize` | `sdo.evaluate` | `sdo.SystemLoggingInfo` | `createSimulator` | `sdo.Experiment`

Topics
“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)”
“Design Optimization to Meet Frequency-Domain Requirements (Code)”
Class Attributes
Property Attributes

sdo.SystemLoggingInfo class

Package: sdo

Specify linear system logging information

Syntax

```
sys = sdo.SystemLoggingInfo
sys = sdo.SystemLoggingInfo(Name,Value)
```

Description

Specify linear system logging information. Use `sdo.SystemLoggingInfo` object to set the `SystemLoggingInfo` property of `sdo.SimulationTest`, to specify linear systems to log when simulating the model. You can configure `sdo.SystemLoggingInfo` to compute the linear system with or without using any frequency-domain check blocks defined in the model.

Construction

`sys = sdo.SystemLoggingInfo` constructs an `sdo.SystemLoggingInfo` object, `sys`, with default linear system logging settings. To modify settings for your specific application, use dot notation.

`sys = sdo.SystemLoggingInfo(Name,Value)` specifies additional linear system logging settings, using one or more `Name,Value` pair arguments. `Name` is a property name on page 2-101 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Use `Name,Value` arguments to specify properties on page 2-101 of `sdo.SystemLoggingInfo` object during object creation. For example, `sys = sdo.SystemLoggingInfo('LoggingName','linear_system1')` creates a `sdo.SystemLoggingInfo` object specifying the `LoggingName` property as `linear_system1`.

Properties

`Source` — Specify how to compute the linear system

`''` (default) | character vector

Specify how to compute the linear system, with or without using frequency-domain check blocks in the model, specified as a character vector. To use a frequency-domain check block in the model,

specify `Source` as the full path of the check block. For example, `'sdorectifier/Filter Design Requirements'`. To not use a model check block, set `Source` to a name or use the default value `''`.

`LoggingName` — Name used for the computed linear system
`'sys'` (default) | character vector

Name used for the computed linear system, specified as a character vector. `LoggingName` appears in the `sdo.SimulationTest.LoggedData` when the simulation is run.

`LinearizationIOs` — Linearization input/output points
`[ ]` (default) | vector of linearization IOs

Linearization input/output (IO) points, specified as a vector of linearization IOs. Create `LinearizationIOs` using the `linio` command from Simulink Control Design.

If `Source` is specified as path to a model frequency-domain check block, and `LinearizationIOs` is non-empty, the linearization IO points of the check block are overwritten when the model is simulated.

`SnapshotTimes` — Linearization snapshot times
`[ ]` (default) | scalar | vector of scalars

Linearization snapshot times, specified as a scalar or vector of scalars.

If `Source` is specified as path to a model frequency-domain check block, and `SnapshotTimes` is non-empty, the linearization snapshot times of the check block are overwritten when the model is simulated.

`LinearizationOptions` — Linearization options
`[ ]` (default) | linearization option set

Linearization options to use when computing the linear system, specified as a linearization option set. To set these options, use the `linearizeOptions` command from Simulink Control Design.

If `Source` is specified as path to a model frequency-domain check block, and `LinearizationOptions` is non-empty, the linearization options of the check block are overwritten when the model is simulated.

Copy Semantics

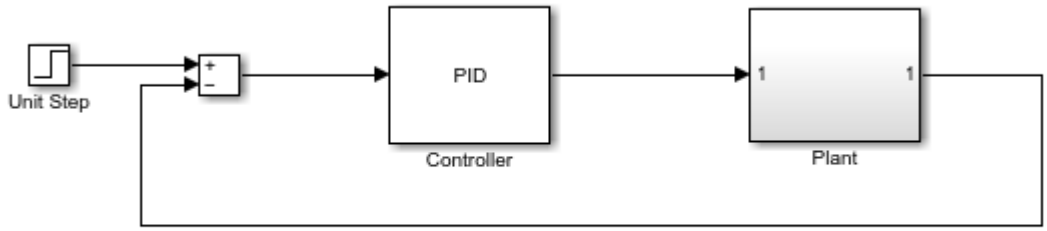
Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Specify Linear System Logging Settings

Open the model.

```
open_system('sldo_model1')
```



Copyright 2008-2021 The MathWorks, Inc.

Specify the input and output points that define the linear system to be computed.

```
IOs(1) = linio('sldo_model1/Sum',1,'input');
IOs(2) = linio('sldo_model1/Plant',1,'output');
```

Create an sdo.SystemLoggingInfo object to specify the linear system logging settings.

```
sys1 = sdo.SystemLoggingInfo;
sys1.Source = 'Specified IOs';
sys1.LoggingName = 'Linear_System';
sys1.LinearizationIOs = IOs;
sys1.SnapshotTimes = 0;
```

See Also

`sdo.SimulationTest` | `linio` | `linearizeOptions` | `fastRestart`

Topics

“Design Optimization to Meet Frequency-Domain Requirements (Code)”
“Use Fast Restart Mode During Response Optimization”
“Use Fast Restart Mode During Sensitivity Analysis”

Introduced in R2015b

3

Methods and Apps

copy

Copy requirement

Syntax

```
copy_req = copy(req)
```

Description

`copy_req = copy(req)` copies a requirement object (`sdo.requirements.StepResponseEnvelope`, ...) to a new object of the same type.

For more information, see `copy`.

Input Arguments

`req`

requirement object (`sdo.requirements.StepResponseEnvelope`, ...)

Output Arguments

`copy_req`

requirement object (`sdo.requirements.StepResponseEnvelope`, ...), which is a copy of `req`.

See Also

`get` | `handle`

Introduced in R2011b

evalRequirement

Class: `sdo.requirements.BodeMagnitude`
Package: `sdo.requirements`

Evaluate Bode magnitude bound for linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` evaluate whether a linear system satisfies the specified piecewise-linear Bode magnitude bound.

Input Arguments

`req`

`sdo.requirements.BodeMagnitude` object.

For MIMO systems, the bound applies to each input/output (I/O) channel.

`lin_sys`

Linear system (`tf`, `ss`, `zpk`, `frd`, `genss`, or `genfrd`).

Output Arguments

`c`

Column vector indicating the maximum signed distance of the system gain to each edge specified in `req`. Negative values indicate that the bound edge is satisfied and positive values that the bound edge is violated.

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.

Examples

Evaluate Bode magnitude requirement.

```
req = sdo.requirements.BodeMagnitude;
sys = tf(1,[1 2 2 1])
c = evalRequirement(req,sys);
```

`c` is negative, which indicates that the system satisfies the gain requirement.

See Also

`sdo.requirements.BodeMagnitude` | `get` | `set` | `copy`

evalRequirement

Evaluate peak gain bound for linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` 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`

- 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.

```
req = sdo.requirements.ClosedLoopPeakGain;
sys = tf(0.5,[1 3 3 1]);
c = evalRequirement(req,sys);
```

`c` is negative, which indicates that the system satisfies the gain requirement.

See Also

`sdo.requirements.ClosedLoopPeakGain` | `get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.FunctionMatching`
Package: `sdo.requirements`

Evaluate satisfaction of function matching requirement

Syntax

```
evaluation = evalRequirement(requirement,dependentVar)
evaluation = evalRequirement(requirement,dependentVar,
indepVar1,...,indepVarN)
```

Description

`evaluation = evalRequirement(requirement,dependentVar)` evaluates whether the test data `dependentVar` matches the function that is specified in the `Type` property of the `requirement` object. The software computes the specified function using default independent variable vectors with value `[0 1 2 ...]`. There is an independent variable vector corresponding to each dimension of `dependentVar`, and the length of each independent variable vector is the same as the size of `dependentVar` in the corresponding dimension.

For example, consider a two-dimensional `dependentVar` of size 3-by-2. To compute a linear function of the form $a_0 + a_1X_1 + a_2X_2$, the software uses the independent variable vectors X_1 `= [0 1 2]` and X_2 `= [0 1]`. The software calculates the fit coefficients a_0, a_1, and a_2 and then calculates the error between the test data and the linear function.

`evaluation = evalRequirement(requirement,dependentVar,
indepVar1,...,indepVarN)` specifies the independent variable vectors to use for computing the function.

Input Arguments

`requirement` — Function matching requirement
`sdo.requirements.FunctionMatching` object

Function matching requirement, specified as an `sdo.requirements.FunctionMatching` object. You specify the function to be matched in `requirement.Type`.

`dependentVar` — Dependent variable test data to be evaluated
vector | matrix | multidimensional array

Dependent variable test data to be evaluated, specified as a vector, matrix, or multidimensional array.

`indepVar1,...,indepVarN` — Independent variable vectors
vectors

Independent variable vectors used for computing the function, specified as real, numeric, monotonic vectors. The independent variable vectors must satisfy the following characteristics:

- The number of independent variables *N* must equal the number of dimensions of the test data.

  For example, use two independent variables when the test data `dependentVar` is a matrix, and use three independent variables when the test data is a three-dimensional array.
- The first independent variable vector specifies coordinates going down test data rows, and the second independent variable vector specifies coordinates going across test data columns. The N^{th} independent variable vector specifies coordinates along the N^{th} dimension of `dependentVar`.
- The number of elements in each independent variable vector must match the size of test data in the corresponding dimension.
- The independent variable vectors must be monotonically increasing or decreasing.

In the requirement object, you can specify centering and scaling of the independent variables using the `Centers` and `Scales` properties. The independent variable vectors specified by you are divided by these `Scales` values after subtracting the `Centers` values. For more information, see the property descriptions on the `sdo.requirements.FunctionMatching` reference page.

You can also specify independent variable vectors using a cell array. The number of elements in the cell array must match the number of dimensions in the test data, `dependentVar`. For example, suppose that `dependentVar` is two-dimensional, you can use either of the following syntaxes:

```
evaluation = evalRequirement(requirement,dependentVar,independentVar1,independentVar2);
evaluation = evalRequirement(requirement,dependentVar,{independentVar1,independentVar2});
```

Output Arguments

`evaluation` — Evaluation of the function matching requirement

scalar | vector | matrix | multidimensional array

Evaluation of the function matching requirement, returned as a scalar, vector, matrix, or array, depending on the value of `requirement.Method`.

`evalRequirement` computes an error signal that is the difference between test data and the specified function of the independent variables. The error signal is then processed further to compute `evaluation`. The value of `evaluation` depends on the error processing method specified in `requirement.Method`.

`requirement.Method`	`evaluation`
`'SSE'`	`evaluation` is returned as a scalar value equal to the sum of squares of the errors. A positive value indicates that the requirement is violated, and `0` value indicates that the requirement is satisfied. The closer `evaluation` is to `0`, the better the match between the function and test data.

requirement.Method	evaluation
'SAE'	evaluation is returned as a scalar value equal to the sum of absolute values of the errors. A positive value indicates that the requirement is violated, and 0 value indicates that the requirement is satisfied. The closer evaluation is to 0, the better the match between the function and test data.
'Residuals'	evaluation is returned as a vector, matrix, or array of the same size as the test data dependentVar. evaluation contains the difference between the test data and the specified function of the independent variables.

Examples

Evaluate Function Matching Requirement for One-Dimensional Variable

Create a requirement object to match one-dimensional variable data to a linear function.

```
Requirement = sdo.requirements.FunctionMatching;
```

Specify the Centers and Scales properties for a one-dimensional variable by using the set command. You specify these properties because their default values are for a two-dimensional variable.

```
set(Requirement,'Centers',0,'Scales',1);
```

Specify test data for the one-dimensional variable.

```
dependentVariable = 0.5+5.*(1:5);
```

Evaluate the requirement.

```
evaluation = evalRequirement(Requirement,dependentVariable)
```

```
evaluation = 5.6798e-30
```

The software computes the linear function using the default independent variable vector [0 1 2 3 4] because you did not specify any independent variable vectors. There is one independent variable because the number of independent variables must equal the number of dimensions of the test data. The size of the independent variable vector equals the size of the test data.

In this example, the processing method has the default value of 'SSE', so evaluation is returned as a scalar value equal to the sum of squares of the errors. evaluation is very close to zero, indicating that the dependentVariable test data almost matches a linear function. Note that machine precision can affect the value of evaluation at such small values.

Match Quadratic Function to Two-Dimensional Variable Data

Create a requirement object, and specify the function to be matched.

```
Requirement = sdo.requirements.FunctionMatching('Type','purequadratic');
```

The object specifies that the variables should match a quadratic function with no cross-terms.

Create 2-dimensional test data for the variable.

```
[X1,X2] = ndgrid((-1:1),(-4:2:4));
dependentVar = X1.^2 + X2.^2;
```

Specify independent variable vectors to compute the quadratic function.

The number of independent variable vectors must equal the dimensionality of the test data. In addition, the independent variable vectors must be monotonic and have the same size as the test data in the corresponding dimension.

```
indepVar1 = (-2:0);
indepVar2 = (-6:2:2);
```

Evaluate if the test data satisfies the requirement.

```
evaluation = evalRequirement(Requirement,dependentVar,indepVar1,indepVar2)
```

```
evaluation = 1.5751e-29
```

The `evalRequirement` command computes an error signal that is the difference between test data and the function of the independent variable vectors. The error signal is further processed to compute `evaluation`, based on the error processing method specified in `Requirement.Method`.

In this example, the processing method has the default value of `'SSE'`, so `evaluation` is returned as a scalar value equal to the sum of squares of the errors. `evaluation` is very close to zero, indicating that the `dependentVariable` test data almost matches a pure quadratic function.

Create test data with cross-terms.

```
dependentVariable2 = X1.^2 + X2.^2 + X1.*X2;
```

Evaluate the requirement for the new test data.

```
evaluation2 = evalRequirement(Requirement,dependentVariable2,indepVar1,indepVar2)
```

```
evaluation2 = 5.3333
```

The output `evaluation2` is greater than `evaluation` and is substantially different from `0`, indicating that `dependentVariable2` does not fit a pure-quadratic function as well as `dependentVariable` fits the function.

See Also

`sdo.requirements.FunctionMatching`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.GainPhaseMargin`
Package: `sdo.requirements`

Evaluate gain and phase margin bounds for linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` 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 `FeedbackSign` property of `req`.

Input Arguments

`req`

`sdo.requirements.GainPhaseMargin` object.

`lin_sys`

Linear system (`tf`, `ss`, `zpk`, `frd`, `genss`, or `genfrd`).

Output Arguments

`c`

- Signed distance of the computed gain and phase margins to the bound if the `Type` property of `req` is >= or ==.

  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 ==, 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 `Type` property of `req` is `'max`. Unstable loops return positive values.

Examples

Evaluate gain and phase margin requirements.

```
req = sdo.requirements.GainPhaseMargin;
sys = tf(0.5,[1 3 3 1]);
c = evalRequirement(req,sys);
```

`c` is negative, which indicates that the system satisfies the gain and phase margin requirement.

See Also

get | set | copy

evalRequirement

Class: `sdo.requirements.MonotonicVariable`
Package: `sdo.requirements`

Evaluate satisfaction of monotonic variable requirement

Syntax

```
evaluation = evalRequirement(requirement,variableData)
```

Description

`evaluation = evalRequirement(requirement,variableData)` evaluates whether the test data, `variableData`, satisfy the monotonic variable requirement that is specified in the `requirement` object. A positive `evaluation` value indicates that the requirement has been violated.

Input Arguments

`requirement` — Monotonic variable requirement

`sdo.requirements.MonotonicVariable` object

Monotonic variable requirement, specified as an `sdo.requirements.MonotonicVariable` object. In the object, you specify the monotonicity required for each dimension of the variable in `requirement.Type`.

`variableData` — Variable data to be evaluated

vector | matrix | array

Variable data to be evaluated, specified as a real numeric vector, matrix, or multidimensional array.

Output Arguments

`evaluation` — Evaluation of the monotonic requirement

column vector

Evaluation of the monotonic requirement, returned as a column vector. The number of elements in `evaluation` is the same as the number of dimensions in `variableData`. A positive value in the vector indicates that the requirement has been violated for the corresponding dimension of the `variableData`. The magnitude of `evaluation` corresponds to the difference between two successive elements that come closest to violating the requirement. For an example, see “Evaluate a Monotonic Variable Requirement” on page 3-12.

Examples

Evaluate a Monotonic Variable Requirement

Create a requirement object with default properties.

```
Requirement = sdo.requirements.MonotonicVariable;
```

Specify the requirement type for a 1-dimensional variable as monotonically decreasing.

```
Requirement.Type = {'>'};
```

Specify test data for a 1-dimensional variable.

```
Data = [20; 15; 25; 26];
```

Evaluate if the test data satisfies the requirement.

```
Evaluation = evalRequirement(Requirement,Data)
```

```
Evaluation = 10
```

Since `Evaluation` is a positive number, it shows that the requirement is violated. To understand the magnitude of `Evaluation`, consider the elements of the test data. While 20>15 satisfies the '>' requirement, 15<25 and 25<26 violate the requirement, resulting in a positive `Evaluation` value. Because the elements 15 and 25 violate the requirement the most, the magnitude of `Evaluation` is 10, the difference between these elements.

Evaluate Monotonic Variable Requirement for a Multidimensional Variable

Create a requirement object, and specify the monotonicity for a 2-dimensional variable.

```
Requirement = sdo.requirements.MonotonicVariable('Type',{'<','>'});
```

The object requires the elements in the first dimension of the variable to be monotonically increasing and the elements in the second dimension to be monotonically decreasing.

Specify 2-dimensional test data for the variable.

```
Data = [10 5; 20 24;30 33];
```

Evaluate if the test data satisfies the requirement.

```
Evaluation = evalRequirement(Requirement,Data)
```

```
Evaluation = 2×1

    -9
     4
```

`Evaluation` is column vector with size corresponding to the dimensions of the test data.

To understand the magnitude of `Evaluation`, consider the elements of the test data along each dimension. For the first dimension of the test data, going down the rows, the software checks the '<' requirement. Since 10<20<30 and 5<24<33 both satisfy the requirement, `Evaluation(1)` is a negative number. Because 24<33 comes closest to violating the requirement, the magnitude of `Evaluation` for this dimension is 9, the difference between these two elements.

For the second dimension of the test data, going across the columns, the software checks, the '>' requirement. While 10>5 satisfies the requirement, 20<24 and 30<33 do not satisfy the requirement.

This results in `Evaluation(2)` being a positive number, indicating that the requirement is not satisfied. Because the elements 20 and 24 violate the requirement the most, the magnitude of `Evaluation(2)` is 4, the difference between these elements.

See Also

`sdo.requirements.MonotonicVariable`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.OpenLoopGainPhase`
Package: `sdo.requirements`

Evaluate gain and phase bounds on Nichols response of linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` 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.

```
req = sdo.requirements.OpenLoopGainPhase;
sys = tf(0.5,[1 3 3 1]);
c = evalRequirement(req,sys);
```

See Also

`sdo.requirements.OpenLoopGainPhase` | `get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.PhasePlaneEllipse`
Package: `sdo.requirements`

Evaluate satisfaction of elliptical bound on phase plane trajectory of two signals

Syntax

```
evaluation = evalRequirement(requirement,signal1,signal2)
evaluation = evalRequirement(requirement,signals)
```

Description

`evaluation = evalRequirement(requirement,signal1,signal2)` evaluates whether the phase plane trajectory of the two signals specified in `signal1` and `signal2` satisfies the elliptical bound specified in the `requirement` object. The phase plane trajectory is a plot of the two signals against each other. A positive `evaluation` value indicates that the requirement has been violated.

`evaluation = evalRequirement(requirement,signals)` specifies the two signals as an *n*-by-2 array. The first column corresponds to the first signal, and the second column corresponds to the second signal. `n` is the number of time points in the signals.

Input Arguments

`requirement` — Phase plane ellipse requirement

`sdo.requirements.PhasePlaneEllipse` object

Phase plane ellipse requirement, specified as an `sdo.requirements.PhasePlaneEllipse` object. In the object, you specify the characteristics of the bounding ellipse such as the radii, center, and rotation of the ellipse. You also specify whether the ellipse is an upper or lower bound.

`signal1,signal2` — Signals to be evaluated

two comma-separated `timeseries` objects

Signals to be evaluated, specified as two comma-separated `timeseries` objects.

`signal1` corresponds to the `x`-value of the phase plane trajectory, and `signal2` corresponds to the `y`-value of the phase plane trajectory. For an example, see “Evaluate Elliptical Bound on Phase Plane Trajectory” on page 3-17.

`signals` — Signals to be evaluated

n-by-2 array

Signals to be evaluated, specified as an *n*-by-2 array. The first column corresponds to the first signal, the `x`-value of the phase plane trajectory. The second column corresponds to the second signal, the `y`-value of the phase plane trajectory. `n` is the number of time points in the signals. For an example, see “Evaluate Circular Bound on Phase Plane Trajectory” on page 3-18.

Output Arguments

`evaluation` — Evaluation of the phase plane ellipse requirement

scalar | column vector

Evaluation of the phase plane ellipse requirement, returned as a scalar or column vector depending on the `Method` property of `requirement`. The `evalRequirement` command computes the signed minimum distance of each point in the phase plane trajectory to the bounding ellipse.

- If `requirement.Method` is `'Maximum'`, `evaluation` is a scalar that is the maximum of the signed distances. A positive value indicates that the requirement has been violated and at least one of the trajectory points lies outside the bounding region.
- If `requirement.Method` is `'Residuals'`, `evaluation` is a column vector that contains the signed distances of each point in the phase plane trajectory to the bounding ellipse. A positive value in the vector indicates that the requirement has been violated for that time point. A negative value or zero indicates that the requirement has been satisfied for that time point.

Examples

Evaluate Elliptical Bound on Phase Plane Trajectory

Create a default requirement object.

```
Requirement = sdo.requirements.PhasePlaneEllipse;
```

The requirement object specifies the bounding ellipse as an upper bound with center located at [0,0], and no rotation. The x-axis radius of the ellipse is 1 and a y-axis radius is 0.5.

Specify the test signal data as `timeseries` objects.

```
signal1 = timeseries(1-exp(-(0:10)'));
signal2 = timeseries(sin((0:10)'));
```

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,signal1,signal2)
```

```
Evaluation = 0.6997
```

The default method for requirement evaluation, `Requirement.Method` is `'Maximum'`. Thus, the `evalRequirement` command computes the signed minimum distance of each point in the phase plane trajectory to the bounding ellipse and then returns the maximum of these distances. A positive value indicates that the at least one point on the phase trajectory lies outside the ellipse and violates the requirement.

To see the signed distance of each of the points on the trajectory to the phase plane ellipse, specify the method for evaluation as `'Residuals'`.

```
Requirement.Method = 'Residuals';
```

Evaluate the requirement using the new evaluation method.

```
Evaluation2 = evalRequirement(Requirement,signal1,signal2)
```

```
Evaluation2 = 11×1

   -0.5000
    0.4291
    0.5711
   -0.0078
    0.4850
    0.6695
    0.1133
    0.4079
    0.6997
    0.2102
      ⋮
```

`Evaluation2` is the minimum distance of the phase plane trajectory to the bounding ellipse and is a column vector with length equal to the length of the signals.

A negative value indicates that the phase plane trajectory of the two signals at the corresponding time point lies inside the ellipse and satisfies the requirement. A positive value indicates that the phase plane trajectory of the two signals at the corresponding time point lies outside the ellipse and violates the requirement.

You can see that the maximum value in `Evaluation2` is the same as `Evaluation`.

Evaluate Circular Bound on Phase Plane Trajectory

Create a default requirement object.

```
Requirement = sdo.requirements.PhasePlaneEllipse;
```

Specify a circular bound of radius 2. The bounding circle is an upper bound.

```
Requirement.Radius = [2 2];
```

Specify the test signal data.

```
Signals = [1 2 3 4 5 6; 10 20 30 40 50 60]';
```

The test data is specified as an n-by-2 array. The first column corresponds to the first signal, the x-value of the phase plane trajectory. The second column corresponds to the second signal, the y-value of the phase plane trajectory.

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,Signals)
```

```
Evaluation =

   58.2993
```

A positive `Evaluation` value indicates that the requirement is violated.

See Also

`sdo.requirements.PhasePlaneEllipse`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.PhasePlaneRegion`
Package: `sdo.requirements`

Evaluate satisfaction of piecewise-linear bound on phase plane trajectory of two signals

Syntax

```
evaluation = evalRequirement(requirement,signal1,signal2)
evaluation = evalRequirement(requirement,signals)
```

Description

`evaluation = evalRequirement(requirement,signal1,signal2)` evaluates whether the phase plane trajectory of the two signals `signal1` and `signal2` satisfies the piecewise-linear bound specified in the `requirement` object. The phase plane trajectory is a plot of the two signals against each other. A positive `evaluation` value indicates that the requirement has been violated.

`evaluation = evalRequirement(requirement,signals)` specifies the two signals as an *n*-by-2 array. The first column corresponds to the first signal, and the second column corresponds to the second signal. `n` is the number of time points in the signals.

Input Arguments

`requirement` — Phase plane region requirement

`sdo.requirements.PhasePlaneRegion` object

Phase plane region requirement, specified as an `sdo.requirements.PhasePlaneRegion` object. In the object, you specify the piecewise-linear bounding edges.

`signal1,signal2` — Signals to be evaluated

`timeseries` objects

Signals to be evaluated, specified as `timeseries` objects.

`signal1` corresponds to the `x`-value of the phase plane trajectory, and `signal2` corresponds to the `y`-value of the phase plane trajectory.

`signals` — Signals to be evaluated

n-by-2 array

Signals to be evaluated, specified as an *n*-by-2 array. The first column corresponds to the first signal, the `x`-value of the phase plane trajectory. The second column corresponds to the second signal, the `y`-value of the phase plane trajectory. `n` is the number of time points in the signals.

Output Arguments

`evaluation` — Evaluation of the phase plane region requirement

scalar

Evaluation of the phase plane region requirement, returned as a scalar value. The software finds the trajectory point that is closest to the bounding region and then calculates `evaluation` as a scaled distance between this point and the closest bound edge. A positive value indicates that the requirement has been violated and some or all the trajectory points defined by the two test signals lie outside the specified bounding region. A negative value or `0` indicates that the requirement has been satisfied. When `evaluation` is `0`, the closest trajectory point lies on the edge.

Examples

Evaluate Piecewise-Linear Bound on Phase Plane Trajectory

Create a default requirement object.

```
Requirement = sdo.requirements.PhasePlaneRegion;
```

The requirement object specifies the bounding region as a single edge extending from (-1,0) to (-1,1) in the phase plane defined by two signals. `Requirement.Type` has the default value of `'<='`. This value implies the area to the left of the edge is out of bounds, where the forward direction is the direction of creation of the edge.

Specify the test signal data as `timeseries` objects.

```
signal1 = timeseries(1-exp(-(0:10)'));
signal2 = timeseries(sin((0:10)'));
```

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,signal1,signal2)
```

```
Evaluation = -0.2632
```

A negative value indicates that the requirement is satisfied, and the phase plane trajectory of the two test signal data is in the bounding region.

Specify Bounding Region With Multiple Edges and Evaluate the Requirement

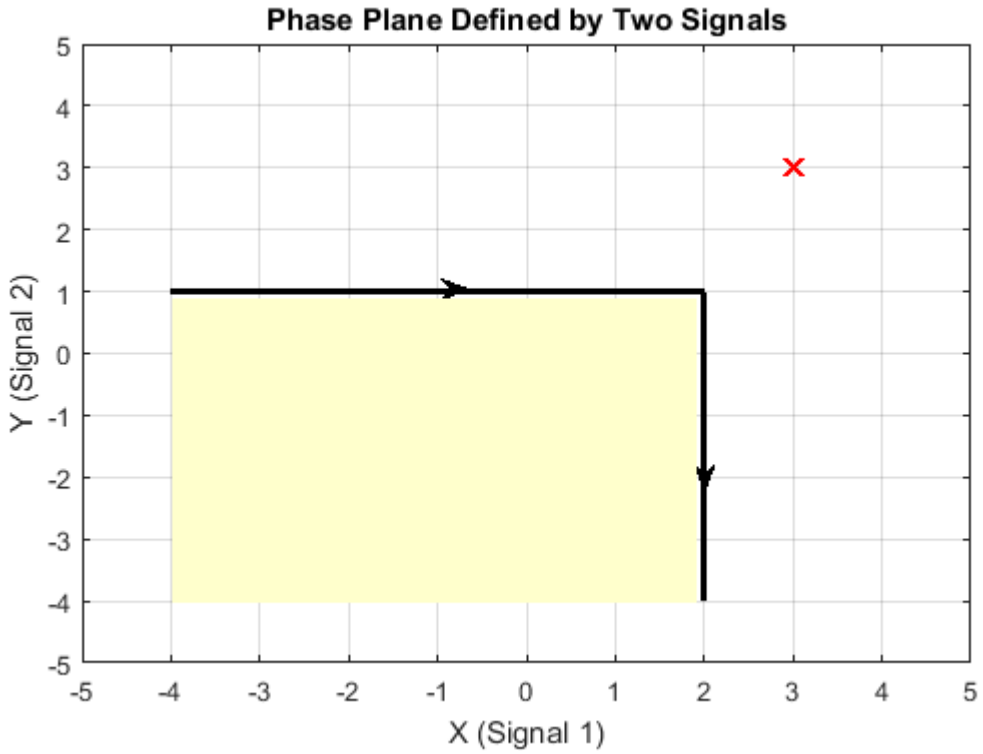
Create a requirement object to specify a piecewise-linear bound on the phase plane trajectory of two signals. The bound has two edges. The first edge extends from (-4,1) to (2,1). The second edge extends from (2,1) to (2,-4).

```
Requirement = sdo.requirements.PhasePlaneRegion('BoundX',[-4 2; 2 2],...
    'BoundY',[1 1; 1 -4]);
```

Specify the bound type as `'>='`.

```
Requirement.Type = '>=';
```

The plot below shows the bounding edges in black. The arrows indicate the direction in which the edges were specified. When you specify the `Type` property as `'>='`, the out-of-bound area is always to the right of each edge, where the forward direction is the direction of creation of the edge. As a result, the out-of bound region is the yellow shaded area, and a trajectory point located at (3,3) is in the bounded region.



Evaluate the requirement for the phase plane trajectory point located at (3,3).

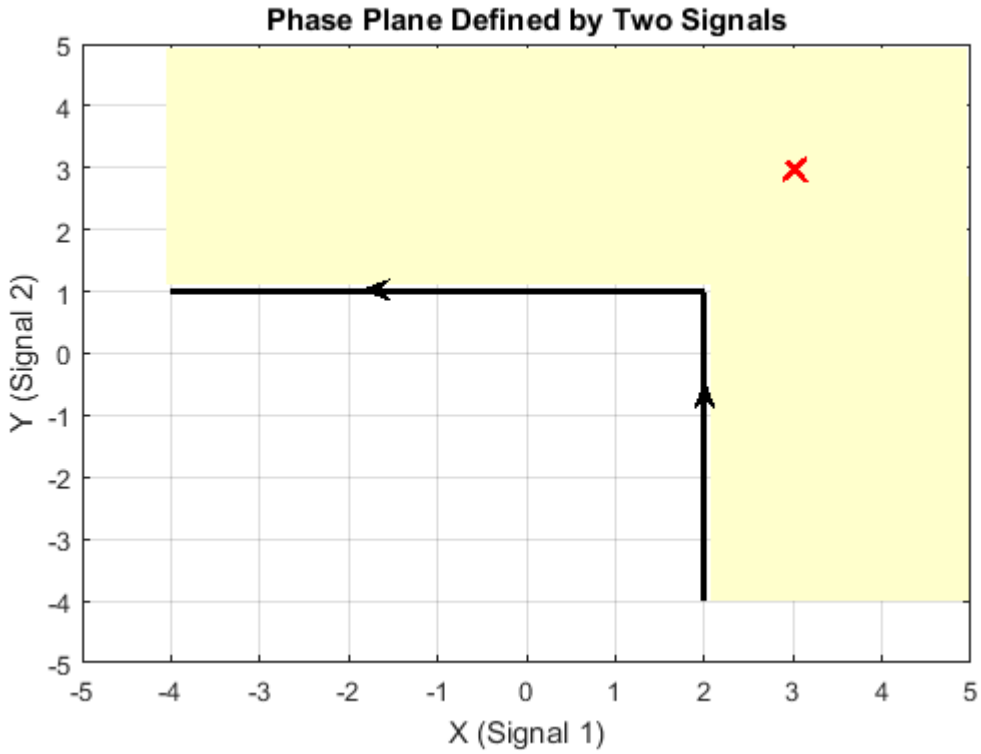
```
Evaluation = evalRequirement(Requirement,[3 3])
```

```
Evaluation = -0.6389
```

`evalRequirement` returns a negative number, indicating the requirement is satisfied.

Now create the requirement by changing the order of specification of the edges.

```
set(Requirement,'BoundX',[2 2; 2 -4],'BoundY',[-4 1;1 1]);
```



The plot shows that the edges were created in the opposite order. So, even though the requirement type is still `'>='`, the out-of-bound region, which is always to the right of the edges, is now flipped.

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,[3 3])
```

```
Evaluation = 0.1087
```

A positive `Evaluation` value indicates the requirement has been violated. Thus, for the same requirement type, the trajectory point at (3,3) is out of bounds when the edges are defined in the reverse order.

See Also

`sdo.requirements.PhasePlaneEllipse`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.PZDampingRatio`
Package: `sdo.requirements`

Evaluate damping ratio bound on linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` 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`

- Signed distance of the damping ratio of each pole of the linear system to the bound, if the `Type` property of `req` is `>=`, `<=` or `==`. Negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. When `==`, 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 `'max'`.

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.

See Also

`get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.PZNaturalFrequency`
Package: `sdo.requirements`

Evaluate natural frequency bound on linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` evaluates whether the poles of a linear system satisfies the specified natural frequency bound.

Input Arguments

`req`

Requirement object (`sdo.requirements.StepResponseEnvelope`, ...).

For MIMO systems, the bound applies to each input/output (I/O) channel.

`lin_sys`

Linear system (`tf`, `ss`, `zpk`, `frd`, `genss`, or `genfrd`).

Output Arguments

`c`

- Signed distance of the natural frequency of each system pole to the bound. If the `Type` property of `req` is `>=`, `<=`, negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. If `==`, 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 `Type` property of `req` is `'max'`.

Examples

Evaluate natural frequency requirement.

```
req = sdo.requirements.PZNaturalFrequency;
sys = tf(0.5,[1 3 3 1]);
c = evalRequirement(req,sys);
```

`c` is positive, which indicates that the system does not satisfy the natural frequency requirement.

See Also

`sdo.requirements.PZNaturalFrequency` | `get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.PZSettlingTime`
Package: `sdo.requirements`

Evaluate settling time bound on linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` 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`

- Signed distance of the real component of each system pole to the bound, if the `Type` property of `req` is `<=` or `==`. Negative values indicate that the bound is satisfied while positive values indicate that the bound is violated. If `==`, values other than 0 indicate that the bound is violated.
- Pole locations such that minimizing the values minimizes the settling time, if the `Type` property of `req` is `'min'`.

Examples

Evaluate settling time requirement.

```
req = sdo.requirements.PZSettlingTime;
sys = tf(0.5,[1 3 3 1]);
c = evalRequirement(req,sys);
```

`c` is positive, which indicates that the system does not satisfy the settling time requirement.

See Also

`get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.RelationalConstraint`
Package: `sdo.requirements`

Evaluate satisfaction of relational constraint requirement

Syntax

```
evaluation = evalRequirement(requirement,variableData1,variableData2)
```

Description

`evaluation = evalRequirement(requirement,variableData1,variableData2)` evaluates whether the test data from two variables, `variableData1` and `variableData2`, satisfy the relational constraint that is specified in the `requirement` object.

Input Arguments

`requirement` — Relational constraint requirement

`sdo.requirements.RelationalConstraint` object

Relational constraint requirement, specified as an `sdo.requirements.RelationalConstraint` object. In the object, you specify the relation required between the elements of two variables.

`variableData1,variableData2` — Variable data to be evaluated

real numeric vectors | real numeric arrays

Variable data from the two variables to be evaluated, specified as real numeric vectors or arrays. The test data of the two variables must be the same size.

Output Arguments

`evaluation` — Evaluation of the relational constraint requirement

vector | array

Evaluation of the relational constraint requirement, returned as a vector or array of the same size as the dimensions of the test data `variableData1`. Note that size of `variableData1` and `variableData2` is the same.

Each element in `evaluation` indicates whether the corresponding elements in `variableData1` and `variableData2` satisfy the requirement. The value returned for each element of `evaluation` depends on the relation specified in the requirement object:

requirement.Type	evaluation Value When Requirement is Satisfied	evaluation Value When Requirement is Violated
'>' or '<'	Negative number with magnitude `|v1-v2|`, the absolute value of difference between the elements `v1` and `v2` of `variableData1` and `variableData2`.	Positive number with magnitude `|v1-v2|`, or `0` if the elements are equal.
'>=' or '<='	Negative number with magnitude `|v1-v2|`, or `0` if the elements are equal.	Positive number with magnitude `|v1-v2|`.
'=='	`0`	Non-zero number that is the difference between the two elements, `v1-v2`.
'~='	`0`	`1`

Examples

Evaluate a Relational Constraint Requirement

Create a requirement object, and specify that the elements of the first variable should be greater than elements of the second variable.

```
Requirement = sdo.requirements.RelationalConstraint('Type','>');
```

Specify test data for the two variables. The test data for both variables must be the same size.

```
varData1 = [20 -3 7];
varData2 = [20 -1 6];
```

Evaluate whether the test data satisfy the requirement.

```
Evaluation = evalRequirement(Requirement,varData1,varData2)
```

```
Evaluation = 1×3

    0.0000    2.0000   -1.0000
```

`Evaluation` is always the same size as the test data. When the relation type is specified as '>', if the requirement is satisfied, `evalRequirement` returns a negative number with magnitude equal to the absolute value of difference between the two elements. If the requirement is violated, `Evaluation` is a positive number with magnitude equal to the absolute value of the difference between the two elements, or `0` if the elements are equal.

The first elements of the two variables are equal, so the requirement is violated and `Evaluation(1)` is `0`, the difference between the elements.

The second elements, -3 and -1, violate the requirement, resulting in a positive `Evaluation(2)` with value = abs(-3-(-1))= 2.

The third elements, 7 and 6, satisfy the requirement, resulting in a negative `Evaluation(3)` with value = -abs(7-6)= -1.

Evaluate Equality of Two Variables

Create a requirement object, and specify that the elements of two variables should be equal to each other.

```
Requirement = sdo.requirements.RelationalConstraint('Type','==');
```

Specify test data for the two variables.

```
varData1 = [20 15];
varData2 = [20 55];
```

Evaluate whether the test data satisfies the requirement.

```
Evaluation = evalRequirement(Requirement,varData1,varData2)
```

```
Evaluation = 1×2

     0   -40
```

`Evaluation` is the same size as the test data. When the relation type is specified as '==', if the requirement is satisfied, `evalRequirement` returns `0`, the difference between the elements. If the requirement is violated, `Evaluation` is a non-zero number equal to the difference between the two elements.

The first elements of the two variables are equal, so the requirement is satisfied and `Evaluation(1)` is `0`.

The second elements, 15 and 55, violate the requirement, resulting in a non-zero `Evaluation(2)`.

See Also

`sdo.requirements.RelationalConstraint`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.SignalBound`
Package: `sdo.requirements`

Evaluate piecewise-linear bound

Syntax

```
c = evalRequirement(req,sig)
```

Description

`c = evalRequirement(req,sig)` 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.

```
req = sdo.requirements.SignalBound;
sig = timeseries(1-exp(-(0:10)'));
c = evalRequirement(req,sig);
```

`c` is negative, which indicates that the signal satisfies the bounds.

See Also

`get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.SignalTracking`
Package: `sdo.requirements`

Evaluate tracking requirement

Syntax

```
c = evalRequirement(req,sig)
c = evalRequirement(req,sig,ref)
```

Description

`c = evalRequirement(req,sig)` evaluates whether a test point signal, `sig`, tracks the reference signal specified by a requirement object, `req`.

`c = evalRequirement(req,sig,ref)` evaluates whether `sig` tracks the reference signal specified by `ref`. `req` 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 `sig` and `ref`, and re-use the requirement object, `req`.

Input Arguments

`req`

`sdo.requirements.SignalTracking` object.

`sig`

MATLAB `timeseries` object or `nxm` array, where the 1st column is time and subsequent columns are signal values.

`ref`

Reference signal, specified as a MATLAB `timeseries` object.

Output Arguments

`c`

- Measure of how well the test point signal matches the reference signal, if the `Type` property of `req` is `'=='`. Specify the algorithm used to compute the tracking measure through the `Method` property.
- Signed distance of the test point signal to the reference signal, if the `Type` property of `req` is `'>='` or `'<='`. Negative values indicate the bound is satisfied while positive values indicate that the bound is violated.

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.

Create a reference and test point signal. Then, use a requirement object to evaluate the requirement.

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.

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

evalRequirement

Class: `sdo.requirements.SingularValue`
Package: `sdo.requirements`

Evaluate singular value bound on linear system

Syntax

```
c = evalRequirement(req,lin_sys)
```

Description

`c = evalRequirement(req,lin_sys)` evaluates whether a linear system satisfies the specified singular values bound.

Input Arguments

`req`

`sdo.requirements.SingularValue` object.

For MIMO systems, the bound applies to each input/output (I/O) channel.

`lin_sys`

Linear system (`tf`, `ss`, `zpk`, `frd`, `genss`, or `genfrd`).

Output Arguments

`c`

Column vector indicating the maximum signed distance of the system gain to each edge specified in `req`. Negative values indicate that the bound edge is satisfied and positive values indicate that the bound edge is violated.

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.

Examples

Evaluate singular value requirement.

```
req = sdo.requirements.SingularValue;
sys = tf(1,[1 2 2 1]);
c = evalRequirement(req,sys);
```

`c` is negative, which indicates that the system satisfies the gain requirement.

See Also

`sdo.requirements.SingularValue` | `get` | `set` | `copy`

evalRequirement

Class: `sdo.requirements.SmoothnessConstraint`
Package: `sdo.requirements`

Evaluate satisfaction of smoothness constraint requirement

Syntax

```
evaluation = evalRequirement(requirement,variableData)
evaluation = evalRequirement(requirement,variableData,
indepVar1,...,indepVarN)
```

Description

`evaluation = evalRequirement(requirement,variableData)` evaluates whether the test data, `variableData`, satisfies the smoothness constraint requirement that is specified in the `requirement` object. The software calculates the gradient magnitude of the test data, and compares it to the bound specified in the `GradientBound` property of the object. For calculating the gradient, the software assumes the spacing between data points in each dimension is `1`. A positive `evaluation` value indicates that the requirement has been violated.

For more information about how the gradient magnitude is calculated, see “Algorithms” on page 3-40.

`evaluation = evalRequirement(requirement,variableData, indepVar1,...,indepVarN)` uses `indepVar1,...,indepVarN` to specify the spacing between test data points in each of the `N` dimensions of the data.

Input Arguments

`requirement` — Smoothness constraint requirement

`sdo.requirements.SmoothnessConstraint` object

Smoothness constraint requirement, specified as an `sdo.requirements.SmoothnessConstraint` object. In the object, you specify the upper bound on the gradient magnitude.

`variableData` — Variable data to be evaluated

real numeric vector | real numeric matrix | real numeric array

Variable data to be evaluated, specified as a real numeric vector, matrix, or array.

`indepVar1,...,indepVarN` — Spacing between points of test data in each dimension

scalar | vector

Spacing between points of test data in each of the `N` dimensions of the data, specified as a scalar or vector. The independent variable **`indepVar1`** specifies the spacing going down the test data rows, and **`indepVar2`** specifies spacing along the columns. Similarly, **`indepVarN`** specifies the spacing along the *N*th dimension of test data. You can specify each of `indepVar1,...,indepVarN` as scalars or vectors:

- Scalars — Specify the spacing between test data in the corresponding dimension as a nonzero scalar. For example, suppose that `variableData` is two-dimensional, and the spacing between data in the first dimension is 5 and in the second dimension is 2. Specify **`indepVar1`** as 5 and **`indepVar2`** as 2.
- Vectors — Specify the coordinates of the test data in the corresponding dimension as real, numeric, monotonic vectors. The software uses the coordinates to compute the spacing between test data points in the corresponding dimension. The length of the vector must match the length of the test data in the corresponding dimension. You do not have to specify coordinates with uniform spacing.

  For example, suppose that `variableData` is two-dimensional, and the length of the test data in the first and second dimension is 3 and 5, respectively. The coordinates of the test data in the first dimension are `[1 2 3]`. In the second dimension, the spacing is not uniform and the coordinates of the test data are `[1 2 10 20 30]`. Specify **`indepVar1`** as `[1 2 3]` and **`indepVar2`** as `[1 2 10 20 30]`.

For an example showing the effect of specifying `indepVar1,...,indepVarN`, see “Specify Spacing Between Tests Data Points” on page 3-39.

You can also specify spacing between points of test data using a cell array. The number of elements in the cell array must match the number of dimensions in the test data, `variableData`. For example, suppose that `variableData` is two-dimensional, and the spacing between data in the first dimension is 2. In the second dimension, the coordinates of the test data are `[1 3 6 7 10]`. You can use either of the following syntaxes to specify spacing between test data:

```
evaluation = evalRequirement(requirement,variableData,2,[1 3 6 7 10]);
evaluation = evalRequirement(requirement,variableData,{2,[1 3 6 7 10]});
```

Output Arguments

`evaluation` — Evaluation of the smoothness constraint requirement

scalar

Evaluation of the smoothness constraint requirement, returned as a scalar. A negative value indicates that the bound is satisfied and a positive value indicates that the requirement has been violated.

Examples

Evaluate Smoothness Constraint on a One-Dimensional Variable

Create a requirement object to constrain the gradient magnitude of a one-dimensional variable to 5.5.

```
Requirement = sdo.requirements.SmoothnessConstraint('GradientBound',5);
```

Specify test data for the one-dimensional variable. The data is linear with slope equal to `10`.

```
variableData = 10*(1:5);
```

Evaluate the requirement.

```
Evaluation = evalRequirement(Requirement,variableData)
```

```
Evaluation = 1
```

Since you did not specify the spacing between the test data points, the software computes the gradient magnitude assuming that the spacing is `1`. `Evaluation` is positive, indicating that the test data gradient magnitude is greater than the bound, and the requirement is violated.

Specify Spacing Between Tests Data Points

Create a smoothness constraint requirement object, and specify the gradient bound as 3.

```
Requirement = sdo.requirements.SmoothnessConstraint;
Requirement.GradientBound = 3;
```

Specify test data with gradient magnitude value of 2 for a one-dimensional variable. The spacing between the test points is 2.

```
variableData = 2:2:10;
```

Evaluate the requirement without specifying the spacing between test points. The software assumes a spacing of 1.

```
Evaluation = evalRequirement(Requirement,variableData)
```

```
Evaluation = -0.3333
```

A negative value indicates that the requirement is satisfied.

Evaluate the requirement using 2 as the spacing between test data points.

```
Evaluation = evalRequirement(Requirement,variableData,2)
```

```
Evaluation = -0.6667
```

The increased spacing reduces the gradient value, and the requirement is still satisfied.

Now evaluate the requirement using 0.5 as the spacing.

```
Evaluation = evalRequirement(Requirement,variableData,0.5)
```

```
Evaluation = 0.3333
```

The decreased spacing increases the gradient magnitude value to be above the gradient bound, and the requirement is violated.

Evaluate Smoothness Constraint on a Multidimensional Variable

Create a requirement object to constrain the gradient magnitude of a 2-dimensional variable to be below 5.5.

```
Requirement = sdo.requirements.SmoothnessConstraint;
```

```
Requirement.GradientBound = 5.5;
```

Specify 2-dimensional test data for the variable. In this example, generate test data with a gradient magnitude of `5`, and use `2` as the spacing between the test data points in each dimension.

```
[X1,X2] = ndgrid(1:2:20,1:2:10);
variableData = 3*X1+4*X2;
```

The gradient of the test data in the two dimensions is 3 and 4. Therefore, the gradient magnitude of the test data is 5 ($= \sqrt{3^2 + 4^2}$).

Specify the coordinates of the test data in each dimension.

```
indepVar1 = [1:2:20];
indepVar2 = [1:2:10];
```

The specified coordinates indicate that the spacing between the test data points in both dimensions is 2.

Evaluate if the test data satisfies the requirement.

```
Evaluation = evalRequirement(Requirement,variableData,indepVar1,indepVar2)
```

```
Evaluation = -0.0909
```

`Evaluation` is a negative number, indicating that the requirement is satisfied, and the test data gradient magnitude is below the specified bound.

Algorithms

To understand how the gradient magnitude is computed, consider test data `F` from a two-dimensional variable that is a function of independent variables x_1 and x_2. The gradient is defined as:

$$\nabla F = \frac{\partial F}{\partial x_1}\hat{i} + \frac{\partial F}{\partial x_2}\hat{j}$$

The gradient magnitude is:

$$|\nabla F| = \sqrt{\left(\frac{\partial F}{\partial x_1}\right)^2 + \left(\frac{\partial F}{\partial x_2}\right)^2}$$

Similarly, the gradient for an N-dimensional variable is:

$$|\nabla F| = \sqrt{\left(\frac{\partial F}{\partial x_1}\right)^2 + \left(\frac{\partial F}{\partial x_2}\right)^2 + \cdots + \left(\frac{\partial F}{\partial x_N}\right)^2}$$

To compute the gradient magnitude, the software computes the partial derivative in each dimension by computing the difference between successive test data in that dimension and dividing by the spacing between test data in that dimension. If you specify the spacing between test data in each dimension in `indepVar1,...,indepVarN`, the software uses the specified spacing. If you do not specify the spacing, the software assumes the test data is spaced 1 step apart in each dimension. The software normalizes the final gradient magnitude by the `GradientBound` property of `requirement`, and returns the normalized value in `evaluation`.

See Also

`sdo.requirements.SmoothnessConstraint`

Introduced in R2016b

evalRequirement

Class: `sdo.requirements.StepResponseEnvelope`
Package: `sdo.requirements`

Evaluate satisfaction of step response requirement

Syntax

```
evaluation = evalRequirement(requirement,signal)
```

Description

`evaluation = evalRequirement(requirement,signal)` evaluates whether `signal` satisfies `requirement`, the step response requirement.

Input Arguments

`requirement` — Step response requirement

`sdo.requirements.StepResponseEnvelope` object

Step response requirement, specified as an `sdo.requirements.StepResponseEnvelope` object.

`signal` — Signal to be evaluated

`timeseries` object | matrix | numeric or generalized LTI model

Signal to be evaluated, specified as one of the following:

- `timeseries` object
- Matrix of size `nxm` — Where `n` is the number of time points in the signal and `m` is the number of channels in the signal. The first column is time and subsequent columns are signal values.
- Numeric or generalized linear time invariant (LTI) model — Available with Control System Toolbox™ software.

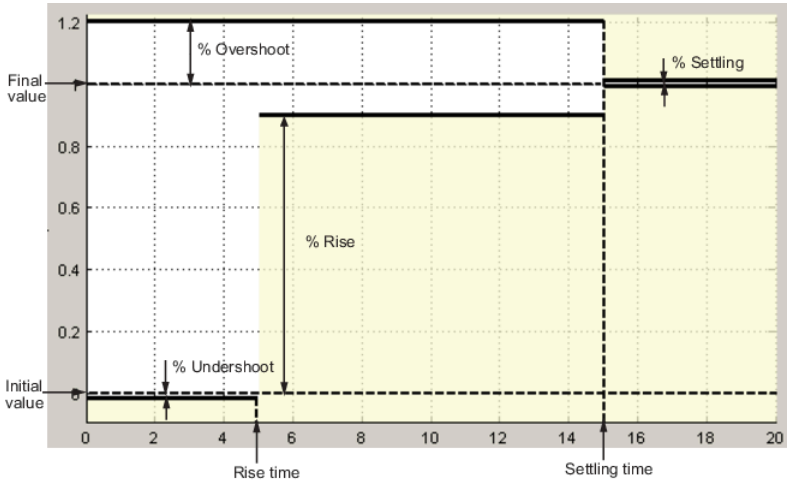
Output Arguments

`evaluation` — Evaluation of the step response requirement

column vector

Evaluation of the step response requirement, returned as a column vector. Negative values indicate that the requirement is satisfied, and positive values indicate that the requirement is violated. `evaluation` consists of the maximum distances between `signal` and step response bounds. The last entry in `evaluation` is a stability measure.

The maximum signal distances from upper bounds are returned before lower bounds. The step bounds are considered in the following order:



- Upper bound for the overshoot — Signal between `StepTime` (time, $t = 0$) and `SettlingTime` of the `requirement` is used.
- Upper bound for the settling time — Signal between `SettlingTime` and 1.5*`SettlingTime` is used.
- Additional upper bound for the settling time — If `signal` extends beyond 1.5*`SettlingTime`, there is an additional upper bound for the settling time. Maximum signal distance from this upper bound is also returned. The signal between 1.5*`SettlingTime` and end of signal is used.
- Lower bound for the undershoot — Signal between `StepTime` and `RiseTime` is used.
- Lower bound for the % rise — Signal between `RiseTime` and `SettlingTime`is used.
- Lower bound for the settling time — Signal between `SettlingTime` and 1.5*`SettlingTime` is used.
- Additional lower bound for the settling time — If `signal` extends beyond 1.5*`SettlingTime`, there is an additional lower bound for the settling time. Maximum signal distance from this lower bound is also returned. The signal between 1.5*`SettlingTime` and end of signal is used.

The last value in `evaluation` is a measure of stability from the end of `signal` to infinity. Negative values indicate that the projected signal is not deviating from the step response after end of `signal`. A positive value indicates that the projected signal is deviating.

Examples

Evaluate Step Response Requirement

Create a step response requirement.

```
requirement = sdo.requirements.StepResponseEnvelope;
```

A requirement is created with the default settling time of 7 seconds.

Specify the signal to be evaluated.

```
signal = timeseries(1-exp(-(0:10)'));
```

The signal data extends to 10 seconds.

Evaluate the step response requirement.

```
evaluation = evalRequirement(requirement,signal)
```

```
evaluation = 6×1

   -0.0917
   -0.0099
   -1.0000
   -0.2416
   -0.0092
   -0.0299
```

The maximum distance of the signal from the step response bounds is returned in `evaluation(1:5)`, followed by the stability value. Negative values indicate that the requirement is satisfied.

The maximum distance from the bounds are returned in the following order:

1. Distance from upper bound for the overshoot
2. Distance from upper bound for the settling time
3. Distance from lower bound for the undershoot
4. Distance from lower bound for the % rise
5. Distance from lower bound for the settling time

`evaluation` does not include the distance of the signal from bounds beyond 1.5*settling time because the signal data does not extend beyond 1.5*settling time.

See Also

`get` | `set` | `copy` | `sdo.requirements.StepResponseEnvelope`

Topics

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

get

Get property values

Syntax

```
get(req)
get(req,PropertyName)
```

Description

`get(req)` returns the value of all properties of the requirement object (`sdo.requirements.StepResponseEnvelope`, ...).

`get(req,PropertyName)` 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 (`sdo.requirements.StepResponseEnvelope`, ...).

`PropertyName`

Name of the requirement object (`sdo.requirements.StepResponseEnvelope`, ...) property.

Alternatives

You can access data in properties using dot notation:

```
object.PropertyName
```

Introduced in R2011b

set

Set property values

Syntax

`set(req,Name,Value,)`

Description

`set(req,Name,Value,)` sets the property value of a requirement object (`sdo.requirements.StepResponseEnvelope`, ...). Specify the property name and value using one or more `Name,Value` pair arguments.

Input Arguments

`req`

Requirement object (`sdo.requirements.StepResponseEnvelope`, ...)

`Name,Value`

Property name of a requirement object (`sdo.requirements.StepResponseEnvelope`, ...), and the corresponding value to set.

Examples

Specify property values.

```
r = sdo.requirements.SignalBound;
set(r,'BoundTimes',[0 5;5 10], ...
    'BoundMagnitudes',[1.1 1.1; 1.01 1.01]);
```

Tips

- Use `set` to simultaneously change properties that you cannot change independently.

Alternatives

You can set data in properties using dot notation: `object.PropertyName`. However, you cannot simultaneously set the value of more than one property using dot notation, use `set` instead.

Introduced in R2011b

getbounds

Get bounds specified in Check block

Syntax

```
bnds = getbounds(blockpath)
```

Description

`bnds = getbounds(blockpath)` returns the bounds specified in the Check block specified by `blockpath`.

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 *model/subsystem/block* 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.

- Check Step Response Characteristics: Cell array of one element that contains a `sdo.requirements.StepResponseEnvelope` 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 `sdo.requirements.SignalBound` objects.
- Check Against Reference: Cell array of one element that contains a `sdo.requirements.SignalTracking` object.

Note Programmatically changing the bound values in the object returned does not update them in the Block Parameters dialog box.

Examples

Get Bounds from Check Block

Retrieve bounds from a Check Step Response Characteristics block.

```
load_system('sldo_model1_stepblk');
allBlkReq = getbounds('sldo_model1_stepblk/Step Response');
```

Type `allBlkReq{1}` to view the cell array element.

```
allBlkReq{1}
```

```
ans =
  StepResponseEnvelope with 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

Topics

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

Introduced in R2011b

getOvershoot

Class: `sdo.requirements.PZDampingRatio`
Package: `sdo.requirements`

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`

setOvershoot

Class: `sdo.requirements.PZDampingRatio`
Package: `sdo.requirements`

Set overshoot to an equivalent damping ratio

Syntax

```
req1 = setOvershoot(req,percent_overshoot)
```

Description

`req1 = setOvershoot(req,percent_overshoot)` 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.

```
req = sdo.requirements.PZDampingRatio
setOvershoot(req,20)
```

See Also

`sdo.requirements.PZDampingRatio` | `evalRequirement` | `getOvershoot`

makedist

Create probability distribution object

Syntax

```
pd = makedist(distname)
pd = makedist(distname,Name,Value)
list = makedist
```

Description

`pd = makedist(distname)` creates a probability distribution object for the distribution `distname`, using the default parameter values.

Use `makedist` to specify uniform, normal, multinomial, piecewise linear, or triangular distribution objects. If you have Statistics and Machine Learning Toolbox software, you can use `makedist` to create objects for other distributions, such as the Gamma or Weibull distributions. For more information, see `makedist`.

To truncate the probability distribution to a specified interval, use `truncate`.

`pd = makedist(distname,Name,Value)` creates a probability distribution object with one or more distribution parameter values specified by name-value pair arguments.

`list = makedist` returns a cell array `list` containing a list of the probability distributions that `makedist` can create.

Examples

Create Normal Distribution Object

Create a normal distribution object using 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);
```

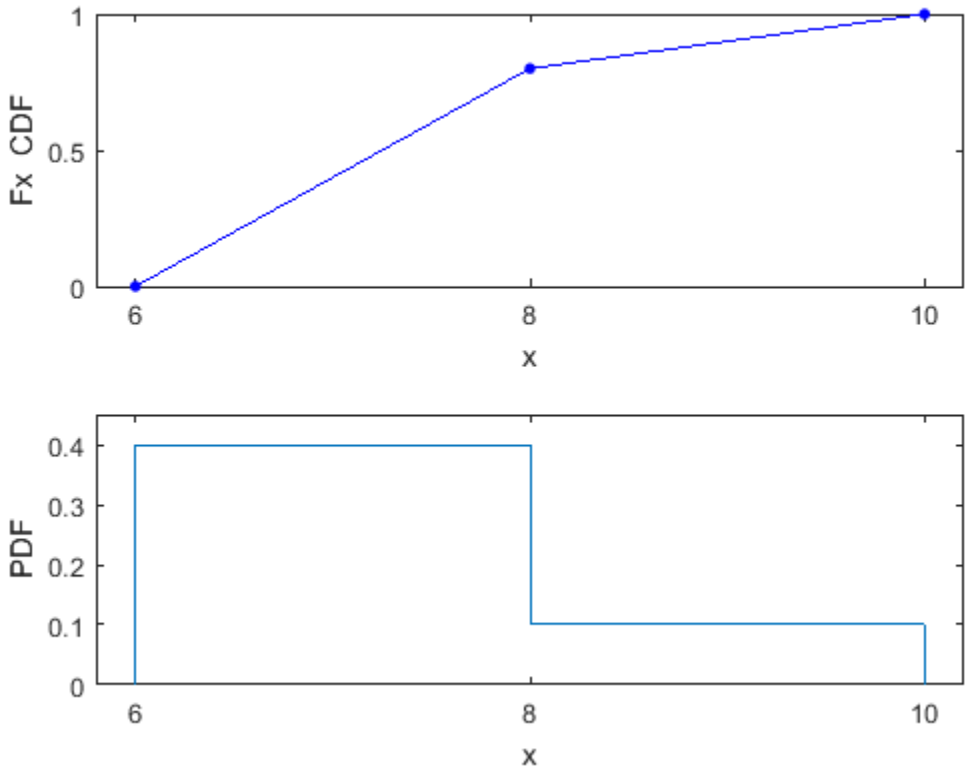
Specify Piecewise Linear Distribution Object

Create a piecewise linear distribution object for a distribution with values from 6 to 10, where values from 6 to 8 are four times more likely than values from 8 to 10.

```
pd = makedist('PieceWiselinear','x',[6 8 10],'Fx',[0 0.8 1]);
```

You specify the cumulative distribution function Fx as [0 0.8 1] because the difference between 0.8 and 0 is four times the difference between 1 and 0.8. As a result, the generated distribution has four times more values between 6 to 8 than between 8 to 10.

The plot shows the specified cumulative distribution function (CDF) and the corresponding probability distribution function (PDF).



The piecewise linear CDF corresponds to a piecewise constant PDF.

Input Arguments

`distname` — Distribution name

`'Uniform'` | `'Normal'` | `'Multinomial'` | `'PiecewiseLinear'` | `'Triangular'`

Distribution name, specified as one of the following values:

Distribution Name	Description
`'Uniform'`	Uniform distribution — You specify the lower and upper bounds of the distribution.
`'Normal'`	Normal distribution — You specify the mean and standard deviation of the distribution.
`'Multinomial'`	Multinomial distribution — In a multinomial distribution, the outcome is one of 1, 2, ..., k. You specify the probability of each outcome, [*p1*,*p2*,..., *pk*]. The probabilities should sum to 1.
`'PiecewiseLinear'`	Piecewise Linear distribution — Use this distribution to make a custom distribution by specifying a piecewise linear cumulative distribution function (CDF). You specify the vector of values, x, at which the CDF changes slope and the corresponding vector of CDF values, Fx. The Fx value at any x is the probability of getting a value less than or equal to x. A piecewise linear CDF is created by linearly connecting the known CDF values, Fx. This piecewise linear CDF corresponds to a piecewise constant probability distribution function. For an example, see “Specify Piecewise Linear Distribution Object” on page 3-52.
`'Triangular'`	Triangular distribution — You specify the lower limit, peak location, and upper limit of the distribution.

For information about these distributions, see the “Probability Distributions” (Statistics and Machine Learning Toolbox) category.

Note If you have Statistics and Machine Learning Toolbox software, you can use `makedist` to create objects for other distributions, such as the Gamma or Weibull distributions. For more information, see `makedist`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Multinomial Distribution

`Probabilities` — Outcome probabilities

`[0.500 0.500]` (default) | vector of scalar values in the range [0,1]

Outcome probabilities for a multinomial distribution, specified as a vector of scalar values in the range [0,1]. The probabilities sum to 1 and correspond to outcomes [1, 2, ..., k], where *k* is the number of elements in the probabilities vector. This argument is valid only when `distname` is `'Multinomial'`.

Example: `'Probabilities',[0.1 0.2 0.5 0.2]` gives the probabilities that the outcome is 1, 2, 3, or 4, respectively.

Data Types: `single` | `double`

Normal Distribution

`mu` — Mean

`0` (default) | scalar value

Mean of a normal distribution, specified as a scalar value. This argument is valid only when `distname` is `'Normal'`.

Example: `'mu',2`

Data Types: `single` | `double`

`sigma` — Standard deviation

`1` (default) | nonnegative scalar value

Standard deviation of a normal distribution, specified as a nonnegative scalar value. This argument is valid only when `distname` is `'Normal'`.

Example: `'sigma',2`

Data Types: `single` | `double`

Piecewise Linear Distribution

`x` — Data values

1 (default) | monotonically increasing vector of scalar values

Data values at which the cumulative distribution function (cdf) changes slope for a piecewise linear distribution, specified as a monotonically increasing vector of scalar values. This argument is valid only when `distname` is `'PiecewiseLinear'`.

Example: `'x',[1 2 3]`

Data Types: `single` | `double`

`Fx` — cdf value at each value in `x`

1 (default) | monotonically increasing vector of scalar values that start at 0 and end at 1

cdf value at each value in `x` for a piecewise linear distribution, specified as a monotonically increasing vector of scalar values that start at 0 and end at 1. This argument is valid only when `distname` is `'PiecewiseLinear'`.

Example: `'Fx',[0.2 0.5 1]`

Data Types: `single` | `double`

Triangular Distribution

A — Lower limit
0 (default) | scalar value

Lower limit for a triangular distribution, specified as a scalar value. This argument is valid only when `distname` is `'Triangular'`.

Example: `'A',-2`

Data Types: `single` | `double`

B — Peak location
0.5 (default) | scalar value greater than or equal to A

Peak location for a triangular distribution, specified as a scalar value greater than or equal to `A`. This argument is valid only when `distname` is `'Triangular'`.

Example: `'B',1`

Data Types: `single` | `double`

C — Upper limit
1 (default) | scalar value greater than or equal to B

Upper limit for a triangular distribution, specified as a scalar value greater than or equal to `B`. This argument is valid only when `distname` is `'Triangular'`.

Example: `'C',5`

Data Types: `single` | `double`

Uniform Distribution

Lower — Lower limit
0 (default) | scalar value

Lower limit for a uniform distribution, specified as a scalar value. This argument is valid only when `distname` is `'Uniform'`.

Example: `'Lower',-4`

Data Types: `single` | `double`

Upper — Upper limit
1 (default) | scalar value greater than Lower

Upper limit for a uniform distribution, specified as a scalar value greater than `Lower`. This argument is valid only when `distname` is `'Uniform'`.

Example: `'Upper',2`

Data Types: `single` | `double`

Output Arguments

pd — Probability distribution
probability distribution object

Probability distribution, returned as a probability distribution object of the type specified by `distname`.

`list` — List of probability distributions
cell array of character vectors

List of probability distributions that `makedist` can create, returned as a cell array of character vectors.

See Also

`sdo.ParameterSpace` | `truncate`

Introduced in R2014a

sdo.analyze

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 *Np*-by-*Nc* table containing the pairwise linear correlation coefficients between each pair of columns in the `x` and `y` tables. `x` contains *Ns* samples of *Np* model parameters. `y` contains *Ns* 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.

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);
```

```
Model evaluated at 50 samples.
```

Obtain the pairwise linear correlation coefficients for the parameters and the cost function.

```
r = sdo.analyze(x,y);
```

Specify Analysis Method

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);
```

```
Model evaluated at 50 samples.
```

Create an options object to use `Correlation` Method.

```
opt = sdo.AnalyzeOptions;
opt.Method = 'Correlation';
```

Obtain the pairwise linear correlation coefficients for the parameters and the cost function.

```
r = sdo.analyze(x,y,opt)
```

```
r=2×3 table
             F          Cleq         leq
          ________    ________    ________

    x1      0.9997    -0.9997           1
    x2    -0.35144    0.35324    -0.35244
```

`r` is a structure with table fields, with one table for each type of analysis specified by `opt`.

Input Arguments

x — Model parameter samples

table

Model parameter samples, specified as an *Ns*-by-*Np* table.

Ns is the number of samples, and *Np* 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 *Ns*-by-*Nc* table.

Ns is the number of samples, and *Nc* 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 *Np*-by-*Nc* table. *Np* is the number of parameters, and *Nc* is the number of cost and constraints returned by the cost function.

See Also
`sdo.AnalyzeOptions` | `sdo.evaluate`

Topics
“Identify Key Parameters for Estimation (Code)”
“Analyze Relation Between Parameters and Design Requirements”

Introduced in R2014a

sdo.evaluate

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 based on the parameter space specifications in `params`, per its `ParameterDistributions`, `RankCorrelation`, and `Options` properties. Np is the number of parameters specified in `params`. `fcn` takes sample values and computes model goal values. A model goal could be a cost (objective), constraint, or assessment of difference between experimental data and model simulation. `sdo.evaluate` applies `fcn` to each row of the table of samples. `y` is a table with one column for each model goal output returned by `fcn` and $2Np+1$ rows. Additional evaluation information is returned in `info`.

`[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 model goal (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.

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);
```

```
Model evaluated at 3 samples.
```

The software generates three samples ($2Np + 1$), and evaluates the `sdoExampleCostFunction` cost function for each sample. `Np` is the number of parameters (= 1).

Input Arguments

`fcn` — Function to be evaluated by `sdo.evaluate`

function handle

Cost function to be evaluated by `sdo.evaluate`, specified as a function handle.

The function requires:

- One input argument, which is a vector of `param.Continuous` object.

  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 (objective) evaluated at `p`. `F` is a `1x1` double.
  - `Cleq` — Value of the nonlinear inequality constraint violations evaluated at `p`.

    `Cleq` is a double `mx1` vector, where `m` is the number of nonlinear inequality constraints.
  - `Ceq` — Value of the nonlinear equality constraint violations evaluated at `p`.

    The value is a double `rx1` vector, where `r` is the number of nonlinear equality constraints.
  - `leq` — Value of the linear inequality constraint violations evaluated at `p`.

    `leq` is a double `nx1` vector, where `n` is the number of linear inequality constraints.
  - `eq` — Value of the linear equality constraint violations evaluated at `p`.

    `eq` is a double `sx1` vector or `[]`, where `s` is the number of linear equality constraints.
  - `Log` — Additional optional information from function evaluation. If specified, this is returned in the `Log` field of output `info`.

Note You can use the same function handle `fcn` for sensitivity analysis, response optimization, or parameter estimation. For optimization and estimation, the solver seeks values of `p` that minimize `F` while satisfying constraints `Cleq`, `Ceq`, `leq` and `eq`. For more information, see `sdo.optimize` and “Write 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 for each evaluation.

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 *Ns* rows.

If you specify `param_samples`, *Ns* is equal to the number of rows of `param_samples`. Otherwise, *Ns* is equal to $2Np+1$. *Np* 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 character vectors.

  Each entry of the cell array is one of the following character vectors:

  - `'success'` — Model evaluation was successful
  - `'failure'` — Model evaluation resulted in all NaN results
  - `'error'` — Model evaluation resulted in an error
- `Log` — Additional evaluation information retrieved from the `Log` field of the cost function, `fcn`.
- `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.Stats.EndTime,info.Stats.StartTime)`.

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true` using `sdo.EvaluateOptions`.

For more information, see “Use Parallel Computing for Sensitivity Analysis”.

See Also

`sdo.EvaluateOptions` | `sdo.optimize` | `sdo.sample` | `sdo.analyze` | `sdo.ParameterSpace`

Topics

“Design Exploration Using Parameter Sampling (Code)”
“Identify Key Parameters for Estimation (Code)”
“Write a Cost Function”
“Create Function Handle”

Introduced in R2014a

createSimulator

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 from Experiment

Specify an experiment.

```
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.

```
load_system('sdoRCCircuit');
p = sdo.getParameterFromModel('sdoRCCircuit','C1');
p.Value = 1e-6;
p.Free = false;
experiment = sdo.Experiment('sdoRCCircuit');
experiment.Parameters = p;
```

Create a simulation scenario for the experiment.

```
sim_obj = createSimulator(experiment);
sim_obj.Parameters.Value
```

```
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`

Topics

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

getValuesToEstimate

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`.

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.

```
load_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'
      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
```

`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(1,2)`, the capacitance of the C1 block, is the model parameter specified by `experiment` for estimation.

See Also

`sdo.Experiment` | `setEstimatedValues`

Topics

“Estimate Model Parameters Using Multiple Experiments (Code)”
“Estimate Model Parameters Per Experiment (Code)”

prepareToDeploy

Configure experiment for deployment with Simulink Compiler

Syntax

```
Experiment_out = prepareToDeploy(Experiment)
```

Description

`Experiment_out = prepareToDeploy(Experiment)` configures an experiment that defines the input and output data for the Simulink model specified as the `sdo.Experiment` object `Experiment` for deployment with Simulink Compiler™.

`prepareToDeploy` saves the initial state and input data in `Experiment_out` so that the `sdo.Experiment` object does not need to perform model error checking or configuration when deployed as part of an executable.

Input Arguments

`Experiment` — Input and output data of Simulink model
`sdo.Experiment` object (default)

Input and output data of Simulink model, specified as an `sdo.Experiment` object.

Output Arguments

`Experiment_out` — Experiment configured for deployment with Simulink Compiler
`sdo.Experiment` object (default)

Experiment configured for deployment with Simulink Compiler, returned as an `sdo.Experiment` object.

Examples

Set Up Experiment and Simulation Test Objects for Deployment

For this example, set up your parameter estimation problem using the **Parameter Estimator** app and generate MATLAB code from it. For more information on generating MATLAB code from the app, see “Generate MATLAB Code for Parameter Estimation Problems (GUI)”. Alternatively, you can also set up your estimation problem at the command line.

Next, split the generated MATLAB code just before the estimation objective function is defined. This results in two files − a `run` function and a `setup` function, as described in “Deployed Application of Parameter Estimation”.

In the `setup` function, add the following lines of code at the end to configure the experiment and simulation test objects for deployment and save them to a MAT-file.

```
Experiment_out = prepareToDeploy(Experiement);
Simulator = createSimulator(Experiment_out);
Simulator = prepareToDeploy(Simulator,p);
save ObjectsToDeploy Experiment_out Simulator p
```

In the `run` function, add the following lines of code at the beginning of the function to include the Simulink model in the compiled code and load the objects that were saved in the `setup` function.

```
%#function simulink_model_name.slx
load ObjectsToDeploy Experiment_out Simulator p
```

Next, add the following lines to load the experiment data and update the experiment object. For this example, assume that the experiment data is contained in the first three columns of a Microsoft Excel spreadsheet file named `fname.xls`.

```
d = xlsread(fname);
u = timeseries(d(:,3),d(:,1));
y = timeseries(d(:,2),d(:,1));
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param1',u);
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param2',y);
```

For a detailed example showing how to deploy your parameter estimation problem using the Simulink Compiler, see “Deployed Application of Parameter Estimation”.

See Also

`updateIOData` | `sdo.Experiment`

Topics

“Deployed Application of Parameter Estimation”

Introduced in R2020a

setEstimatedValues

Update experiments with estimated model initial states and parameter values

Syntax

```
experiment = setEstimatedValues(experiment0,parameters)
```

Description

`experiment = setEstimatedValues(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.

```
load_system('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 μ F.

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

`getValuesToEstimate`

Topics

“Estimate Model Parameters Using Multiple Experiments (Code)”
“Estimate Model Parameters Per Experiment (Code)”

updateIOData

Update experiment input and output data

Syntax

```
Experiment_out = updateIOData(Experiment,sigName,data)
Experiment_out = updateIOData(Experiment,sigName1,data1,sigName2,data2,...)
```

Description

`Experiment_out = updateIOData(Experiment,sigName,data)` updates the signal `sigName` of the experiment object `Experiment` with data specified in `data`.

`updateIOData` lets you reuse the experiment definition but with different data. The function is particularly useful for deployed applications where you do not want to redefine the experiment each time you execute the deployed application.

`Experiment_out = updateIOData(Experiment,sigName1,data1,sigName2,data2,...)` updates multiple signals of an experiment object for deployment with Simulink Compiler.

Input Arguments

`Experiment` — Experiment definition for Simulink model
`sdo.Experiment` object (default)

Experiment definition for Simulink model, specified as an `sdo.Experiment` object.

`sigName` — Signal to be updated
string

Signal to be updated, specified as a string. You can specify `sigName` using the full signal block path or by the signal name if the signal names in the Simulink model are unique.

You can specify multiple `sigName` and `data` pairs to update multiple inputs or outputs in one call to `updateIOData`.

`data` — Data for signal update
time series | `vector`

Data for the signal update, specified as a time series or vector.

You can specify multiple `sigName` and `data` pairs to update multiple inputs or outputs in one call to `updateIOData`.

Output Arguments

`Experiment_out` — Experiment with updated input and output data
`sdo.Experiment` object

Experiment with updated input and output data, returned as an `sdo.Experiment` object.

Examples

Set Up Experiment and Simulation Test Objects for Deployment

For this example, set up your parameter estimation problem using the **Parameter Estimator** app and generate MATLAB code from it. For more information on generating MATLAB code from the app, see “Generate MATLAB Code for Parameter Estimation Problems (GUI)”. Alternatively, you can also set up your estimation problem at the command line.

Next, split the generated MATLAB code just before the estimation objective function is defined. This results in two files − a `run` function and a `setup` function, as described in “Deployed Application of Parameter Estimation”.

In the `setup` function, add the following lines of code at the end to configure the experiment and simulation test objects for deployment and save them to a MAT-file.

```
Experiment_out = prepareToDeploy(Experiement);
Simulator = createSimulator(Experiment_out);
Simulator = prepareToDeploy(Simulator,p);
save ObjectsToDeploy Experiment_out Simulator p
```

In the `run` function, add the following lines of code at the beginning of the function to include the Simulink model in the compiled code and load the objects that were saved in the `setup` function.

```
%#function simulink_model_name.slx
load ObjectsToDeploy Experiment_out Simulator p
```

Next, add the following lines to load the experiment data and update the experiment object. For this example, assume that the experiment data is contained in the first three columns of a Microsoft Excel spreadsheet file named `fname.xls`.

```
d = xlsread(fname);
u = timeseries(d(:,3),d(:,1));
y = timeseries(d(:,2),d(:,1));
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param1',u);
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param2',y);
```

For a detailed example showing how to deploy your parameter estimation problem using the Simulink Compiler, see “Deployed Application of Parameter Estimation”.

See Also

`sdo.Experiment` | `prepareToDeploy`

Topics

“Deployed Application of Parameter Estimation”

Introduced in R2020a

sdo.getModelDependencies

Package: sdo

List of model file and path dependencies

Syntax

```
[dirs,files] = sdo.getModelDependencies(modelname)
```

Description

`[dirs,files] = sdo.getModelDependencies(modelname)` returns dependencies of a Simulink model. The dependencies are required for parallel computing of parameter estimation, response optimization, or sensitivity analysis tasks. The model must be open for the dependency analysis.

`sdo.getModelDependencies` may not return a complete list of model dependencies; some dependencies are undetectable. To learn more, see “Dependency Analyzer Scope and Limitations”. If your model has dependencies that are undetected or inaccessible by the parallel pool workers, then add them to the list of model dependencies.

Input Arguments

`modelname`

Simulink model name, specified as a character vector or string. For example, `'sldo_model1'`.

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.

`files`

Cell array of files that are 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 Dependencies Required for Parallel Computing

Copy Simulink model and boiler library to temporary folder.

```
pathToLib = boilerpressure_setup;
```

Add folder to search path and open model.

```
origPath = addpath(pathToLib);
boilerpressure_demo
```

Get model dependencies.

```
[dirs, files] = sdo.getModelDependencies('boilerpressure_demo');
```

The paths listed in `dirs` are the paths to all the file dependencies listed in `files`.

Enable parallel computing and add model dependencies.

```
opts = sdo.OptimizeOptions;
opts.UseParallel = true;
opts.ParallelFileDependencies = files;
```

Add Additional Files to Model File Dependency List

Copy Simulink model and boiler library to temporary folder.

```
pathToLib = boilerpressure_setup;
```

Add folder to search path and open model.

```
origPath = addpath(pathToLib);
boilerpressure_demo
```

Get model dependencies.

```
[dirs, files] = sdo.getModelDependencies('boilerpressure_demo');
```

Append an additional file, `filename.m` located in `'C:\matlab\work\'`

```
files = vertcat(files,'C:\matlab\work\filename.m');
```

Enable parallel computing and add model dependencies.

```
opts = sdo.OptimizeOptions;
opts.UseParallel = true;
opts.ParallelFileDependencies = files;
```

Make Local Paths Accessible to Remote Workers

Using file dependencies is recommended, however, in some cases it can be better to choose path dependencies. For example, if parallel computing is set up on a local multi-core computer, using path dependencies is preferred as using file dependencies creates multiple copies of the dependency files on the local computer. This example shows how to use path dependencies for setting up parallel computing.

Copy Simulink model and boiler library to temporary folder.

```
pathToLib = boilerpressure_setup;
```

Add folder to search path and open model.

```
origPath = addpath(pathToLib);
boilerpressure_demo
```

Get model dependencies.

```
[dirs, files] = sdo.getModelDependencies('boilerpressure_demo');
```

Add undetected path dependencies.

```
dirs = vertcat(dirs,'//hostname/C$/matlab/work');
```

Replace `C:/` with valid network path accessible to remote workers.

```
dirs = regexprep(dirs,'C:/','////hostname//C$//');
```

Enable parallel computing and add model dependencies.

```
opts = sdo.OptimizeOptions;
opts.UseParallel = true;
opts.ParallelPathDependencies = dirs;
```

Tips

- `files` lists the model dependencies, and `dirs` lists the corresponding paths to these dependencies.

  The model dependencies are required during parallel computing and are made accessible to the parallel pool workers by specifying one of the following:

  - File dependencies: the model dependency files are copied to the parallel pool workers.

    Use `files` to set the `ParallelFileDependencies` property of `sdo.OptimizeOptions` to use for parallel computing.
  - Path dependencies: the paths to the model dependencies are specified to the parallel pool workers.

    Use `dirs` to set the `ParallelPathDependencies` property of `sdo.OptimizeOptions` to use for parallel computing.
- Modify `files` and `dirs` to include dependencies that `sdo.getModelDependencies` cannot detect.
- Using file dependencies is recommended, however, in some cases it can be better to choose path dependencies. For example, if parallel computing is set up on a local multi-core computer, using path dependencies is preferred as using file dependencies creates multiple copies of the dependency files on the local computer.

Alternatives

- “Use Parallel Computing for Parameter Estimation”
- “Use Parallel Computing for Response Optimization”
- “Use Parallel Computing for Sensitivity Analysis”

See Also

`sdo.optimize` | `sdo.OptimizeOptions` | `sdo.evaluate` | `sdo.EvaluateOptions`

Topics
“Improving Optimization Performance Using Parallel Computing”
“Speed Up Response Optimization Using Parallel Computing”
“Speed Up Parameter Estimation Using Parallel Computing”
“Analyze Model Dependencies”

Introduced in R2011b

sdo.getParameterFromModel

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, specified as a character vector or string. For example, `'sldo_model1'`.

`paramname`

Model parameter name, specified as a character vector or string for one parameter. For multiple parameters, specify as a cell array of character vectors or a string array. If a parameter is in a referenced model, the variable name must include the path.

For instance, if a parameter `Ki` is in a referenced model named `Controller` used in the top-level model, use `p_des = sdo.getParameterFromModel('TopLevelModel','Controller:Ki')`.

If `Ki` is a model argument in a referenced model, provide block path from top-level model as follows, `p_des = sdo.getParameterFromModel('TopLevelModel','TopLevelModel/ControlBlock:Ki')`. Here, `ControlBlock` is the block name in the referenced model.

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

```
load_system('sldo_model1_stepblk');

p_des = sdo.getParameterFromModel('sldo_model1_stepblk','Kp');
```

Get Multiple Model Parameters as Optimization Design Variables

```
paramname = {'Kp','Ki','Kd'};

load_system('sldo_model1_stepblk');

p_des = sdo.getParameterFromModel('sldo_model1_stepblk',paramname);
```

Get All Model Parameters as Optimization Design Variables

```
load_system('sldo_model1_stepblk');
p_des = sdo.getParameterFromModel('sldo_model1_stepblk');
```

Alternatives

“Specify Design Variables”

See Also

`sdo.optimize` | `sdo.setValueInModel`

Topics

“Design Optimization to Meet Step Response Requirements (Code)”
“Estimate Model Parameter Values (Code)”
“Design Optimization Tuning Parameters in Referenced Models (Code)”

Introduced in R2011b

sdo.getStateFromModel

Package: sdo

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 character vector or string. For example, `'sdoAircraft'`.

The model must be open.

`blockpath`

Block path of the block containing the required state, specified as a character vector or string. For example, `'sdoAircraft/Actuator Model'`.

To specify multiple blocks, use a cell array of character vectors or a string array.

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

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

Get Model States

```
modelname = 'sdoAircraft';
load_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`

Topics

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

Introduced in R2011b

sdo.getValueFromModel

Package: sdo

Get design variable value from model

Syntax

```
param_value = sdo.getValueFromModel(modelname,param_des)
```

Description

`param_value = sdo.getValueFromModel(modelname,param_des)` gets the value of a design variable in a Simulink model. The model must be open.

Input Arguments

`modelname`

Simulink model name, specified as a character vector or string. For example, `'sldo_model1'`.

`param_des`

Design variables, specified as:

- Character vector for one design variable. For multiple variables, specify as a cell array of character vectors or a string array. For example, `{'Kp','Ki'}`.
- A `param.Continuous` object for one variable or a vector of objects for multiple variables, created using `sdo.getParameterFromModel`.

If a parameter is in a referenced model, the variable name must include the path. For instance, if a parameter `Ki` is in a referenced model named `Controller` used in a top-level model, use `p_des = sdo.getValueFromModel('TopLevelModel','Controller:Ki')`.

If `Ki` is a model argument in a referenced model, provide block path from top-level model as follows, `p_des = sdo.getValueFromModel('TopLevelModel','TopLevelModel/ControlBlock:Ki')`. Here, `ControlBlock` is the block name in the referenced model.

Output Arguments

`param_value`

Design variable value in the model.

A cell array for multiple variable values.

Examples

Get Current Design Variable Value From Model

```
load_system('sldo_model1_stepblk');
p_value = sdo.getValueFromModel('sldo_model1_stepblk','Kp');
```

Alternatively, type:

```
p_des = sdo.getParameterFromModel('sldo_model1_stepblk','Kp');
p_value = sdo.getValueFromModel('sldo_model1_stepblk',p_des);
```

See Also

`sdo.optimize`

Topics

“Design Optimization Tuning Parameters in Referenced Models (Code)”

Introduced in R2011b

sdo.scatterPlot

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.

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 `sdoHydraulicCyclinder` 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 `sdoHydraulicCyclinder` 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);
```

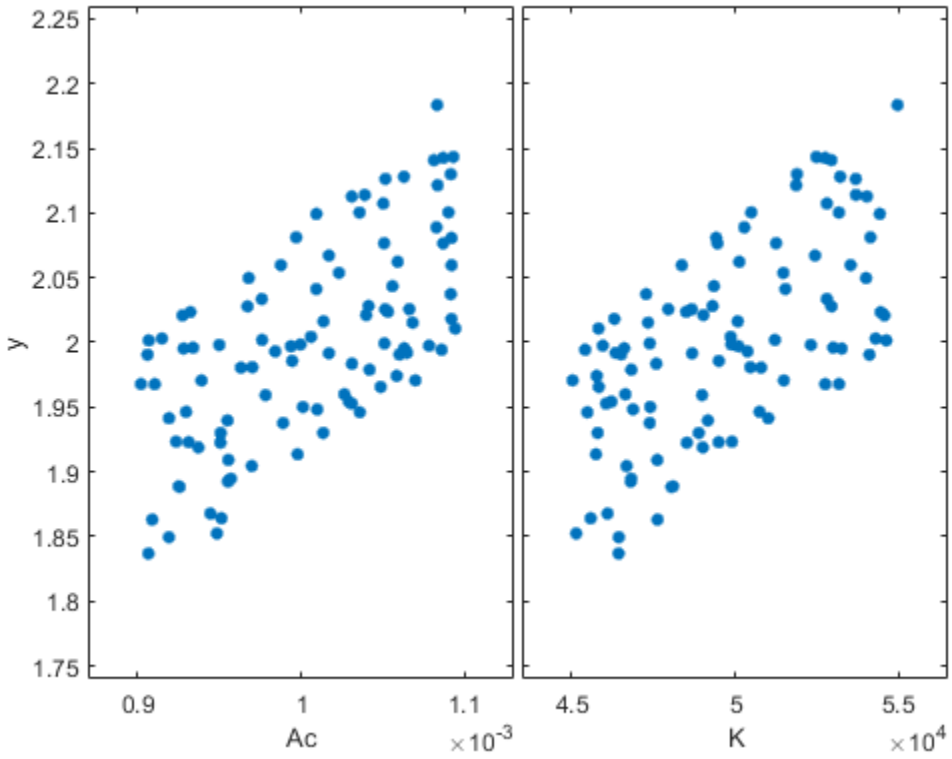
The first operation obtains the `Ac` and `K` parameters as a vector, `p`. The second operation creates an `sdo.ParameterSpace` object, `ps`, that specifies the probability distributions of the parameter samples. The third operation generates 100 samples of each parameter, returned as a `Table`, `X`.

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 `sdoHydraulicCyclinder` model. Use a scatter plot to analyze the samples.

Load the `sdoHydraulicCyclinder` model.

```
load_system('sdoHydraulicCylinder');
```

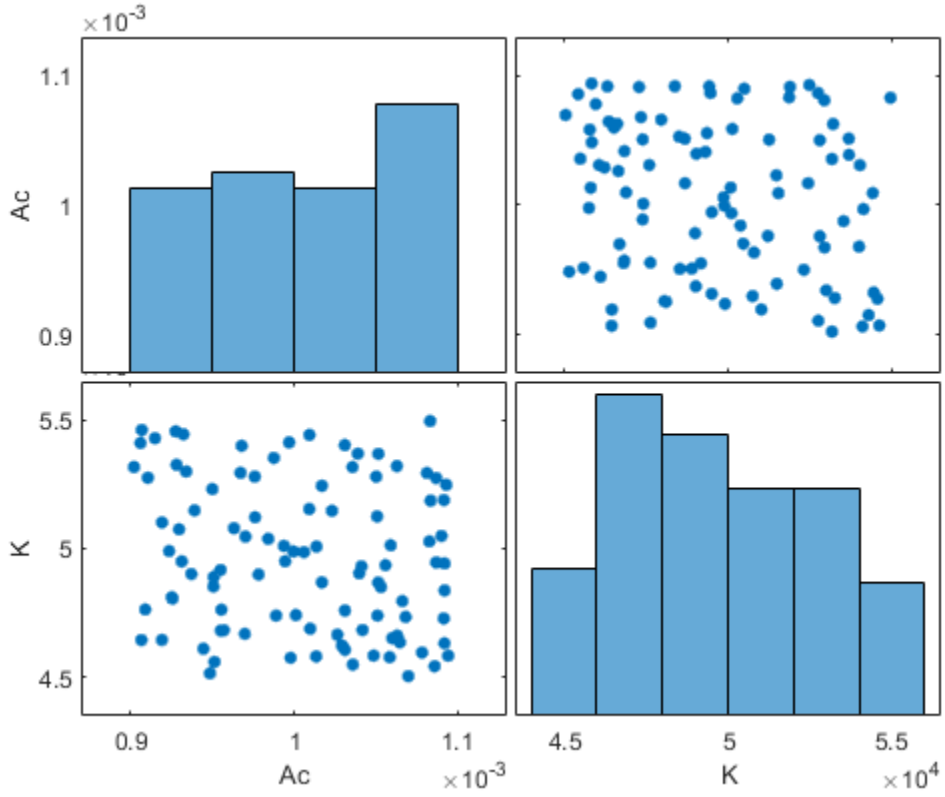
Generate 100 samples of the Ac and K parameters.

```
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);
```



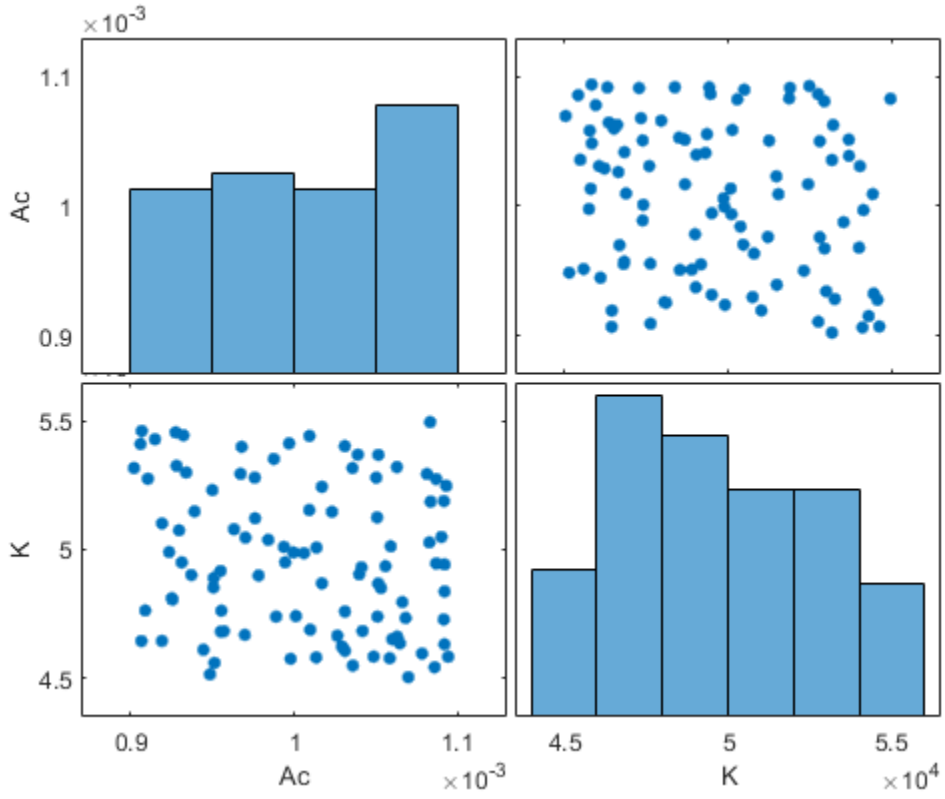
Set Scatter Plot Properties Using Handles

Generate samples of the Ac and K parameters of the sdoHydraulicCyclinder model.

```
load_system('sdoHydraulicCylinder');
p = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
ps = sdo.ParameterSpace(p);
X = sdo.sample(ps,100);
```

Create a scatter plot matrix and return the object handles and the axes handles.

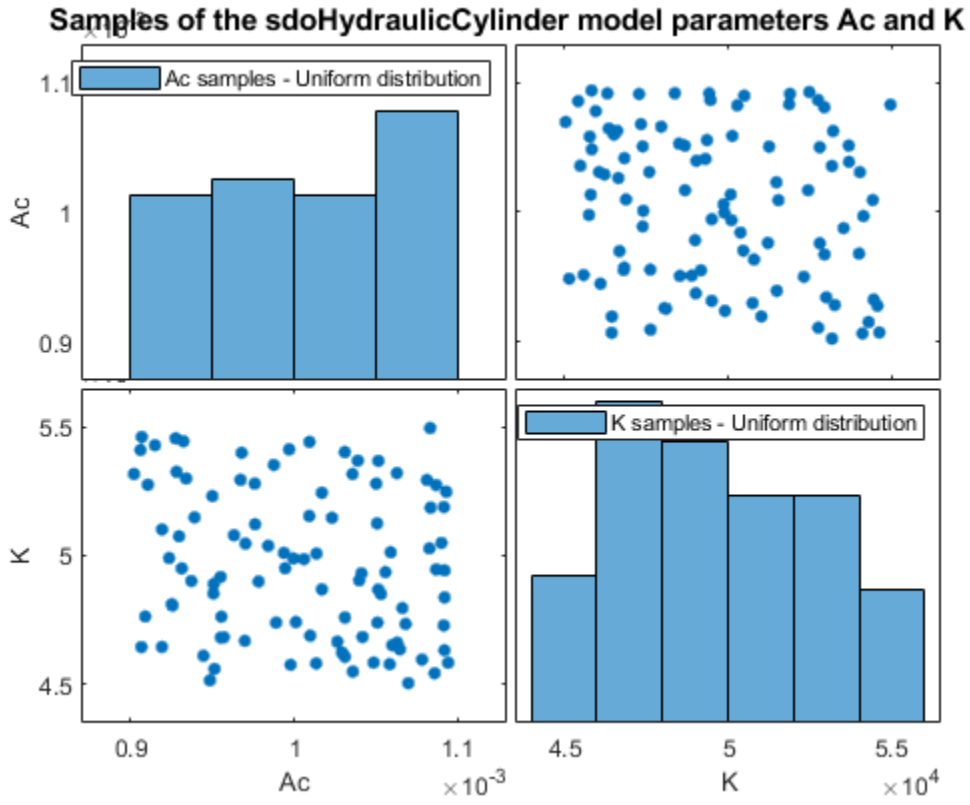
```
figure
[H,AX,BigAX,P,PAx] = sdo.scatterPlot(X);
```



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(PAx(1),'Ac samples - Uniform distribution')
legend(PAx(2),'K samples - Uniform distribution')
```



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.

See Also

`sdo.sample` | `sdo.evaluate`

Topics
“Design Exploration Using Parameter Sampling (Code)”
“Identify Key Parameters for Estimation (Code)”
“Analyze Relation Between Parameters and Design Requirements”

Introduced in R2014a

isreal

Determine if parameter value, minimum, and maximum are real

Syntax

```
isreal(param_obj)
```

Description

`isreal(param_obj)` returns `true (1)` if the `Value`, `Minimum` and `Maximum` properties of `param_obj` are all real.

Input Arguments

param_obj

A `param.Continuous` object.

Default:

Examples

Determine if Parameter Value and Minimum/Maximum Bounds are Real

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

```
ans = logical
   1
```

Because the `Value`, `Minimum`, and `Maximum` properties of all parameters in `p` are real, `isreal` returns 1.

See Also

`param.Continuous`

Parameter Estimator

Estimate model parameters and initial states

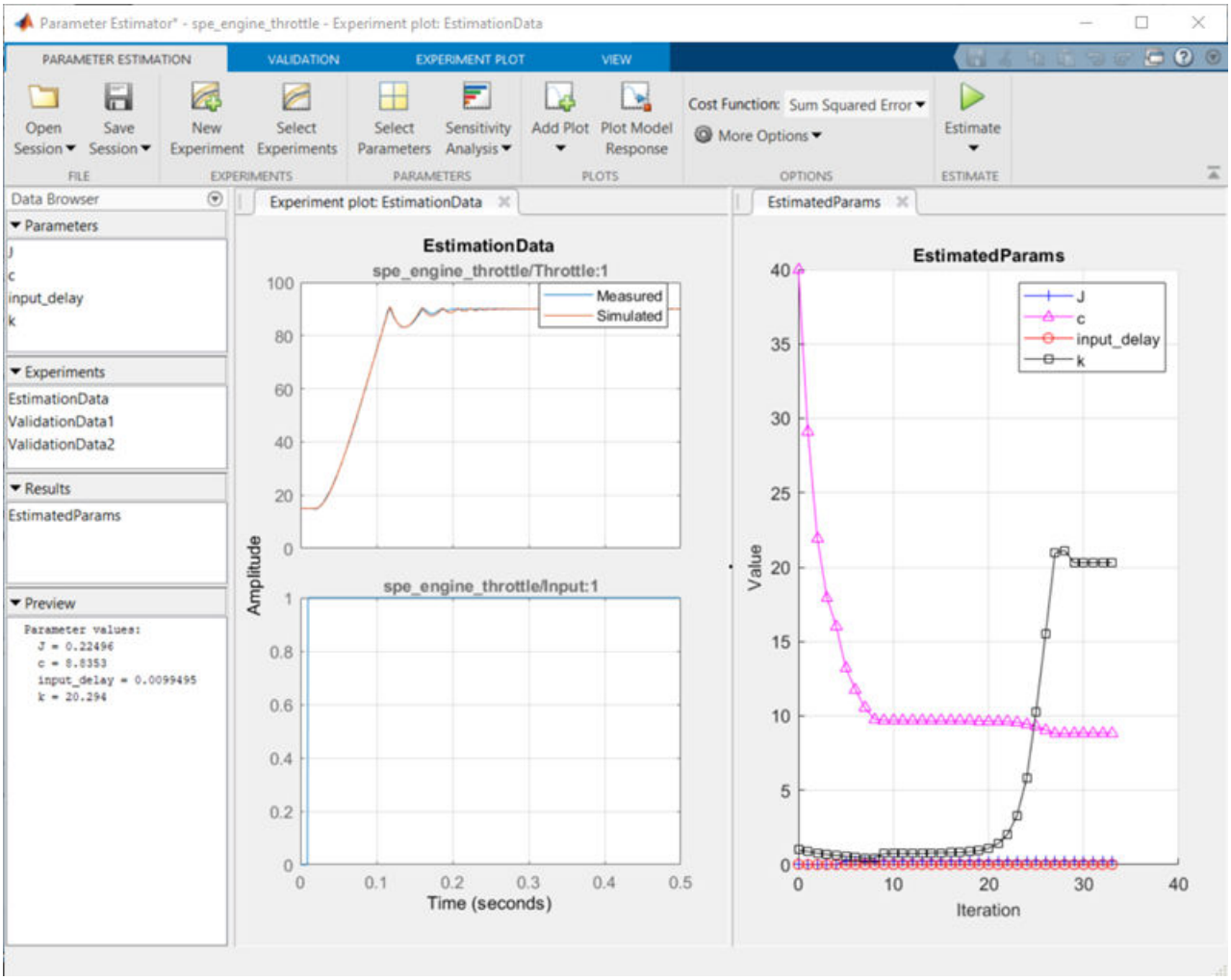
Description

The **Parameter Estimator** app estimates parameters and initial states of a Simulink model using measured data. The app increases model accuracy so that the model reflects the measured hardware behavior. For example, you can automatically estimate electric motor resistance, inductance, and inertia from measured voltage and motor speed data.

Using this app, you can:

- Import and preprocess measured data.
- Find the most influential parameters to estimate (with the **Sensitivity Analyzer**).
- Estimate model parameters and initial states, and monitor estimation progress.
- Validate estimation results.

You can generate MATLAB code from the app, and accelerate parameter estimation using parallel computing and Simulink fast restart.



Open the Parameter Estimator App

- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click **Parameter Estimator**.
- MATLAB command prompt: Enter `spetool('modelname')`.

Examples

- “How the Software Formulates Parameter Estimation as an Optimization Problem”
- “Estimate Parameters from Measured Data”
- “Estimate Model Parameter Values (GUI)”
- “Estimate Model Parameters Per Experiment (GUI)”
- “Estimate Model Parameters and Initial States (GUI)”

Programmatic Use

`spetool('modelname')` opens the **Parameter Estimator** app and creates a new session. The model must be open or on the MATLAB path.

`spetool(spesession)` opens a previously saved **Parameter Estimator** app session.

See Also

`sdo.optimize`

Topics

“How the Software Formulates Parameter Estimation as an Optimization Problem”
“Estimate Parameters from Measured Data”
“Estimate Model Parameter Values (GUI)”
“Estimate Model Parameters Per Experiment (GUI)”
“Estimate Model Parameters and Initial States (GUI)”

Introduced before R2006a

Response Optimizer

Optimize model response to satisfy design requirements, test model robustness

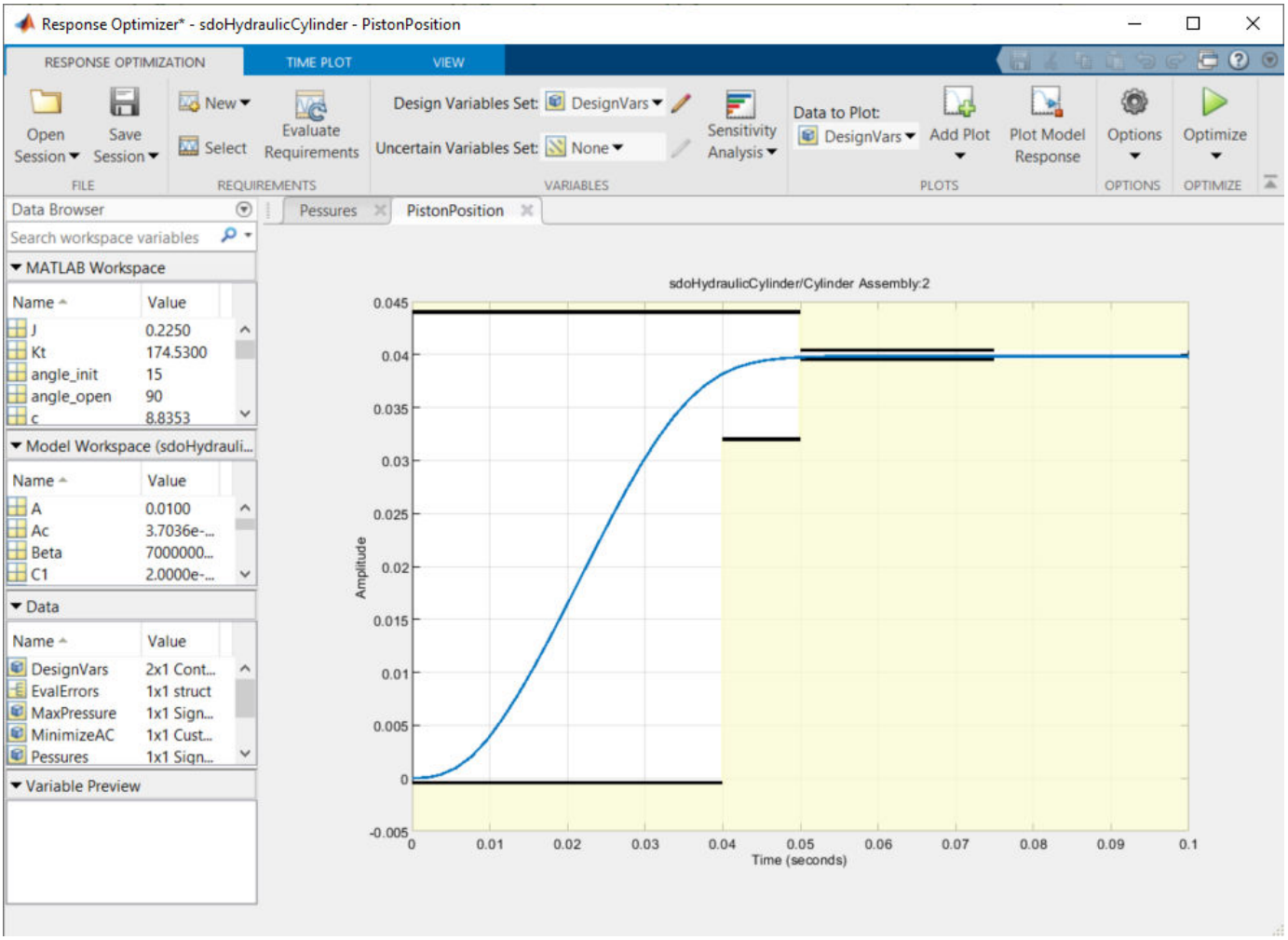
Description

The **Response Optimizer** app automatically optimizes system parameters to improve design characteristics such as response time, bandwidth, and energy consumption. The system parameters can be tuned to meet time-domain and frequency-domain requirements such as overshoot and phase margin, and custom requirements.

Using this app, you can:

- Specify design requirements.
- Incorporate parameter uncertainty to validate the robustness of your design.
- Find the most influential parameters to optimize (with the **Sensitivity Analyzer**).
- Optimize model parameters, and monitor optimization progress.

You can generate MATLAB code from the app, and you can accelerate response optimization using parallel computing and Simulink fast restart.



Open the Response Optimizer App

- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click the **Response Optimizer**.
- MATLAB command prompt: Enter `sdotool('modelname')`.

Examples

- “How the Optimization Algorithm Formulates Minimization Problems”
- “Design Optimization to Meet Step Response Requirements (GUI)”
- “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)”
- “Design Optimization to Track Reference Signal (GUI)”
- “Design Optimization Using Frequency-Domain Check Blocks (GUI)”
- “Design Optimization to Meet a Custom Objective (GUI)”
- “Specify Custom Signal Objective with Uncertain Variable (GUI)”

Programmatic Use

`sdotool('modelname')` opens the **Response Optimizer** app and creates a new session. The model must be open or on the MATLAB path.

`sdotool(sdosession)` opens a previously saved **Response Optimizer** app session.

See Also

`sdo.optimize`

Topics

“How the Optimization Algorithm Formulates Minimization Problems”
“Design Optimization to Meet Step Response Requirements (GUI)”
“Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)”
“Design Optimization to Track Reference Signal (GUI)”
“Design Optimization Using Frequency-Domain Check Blocks (GUI)”
“Design Optimization to Meet a Custom Objective (GUI)”
“Specify Custom Signal Objective with Uncertain Variable (GUI)”

Introduced in R2011b

addParameter

Class: `sdo.ParameterSpace`
Package: `sdo`

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, `ps`, for the `Ac` parameter of `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);
```

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` | `removeParameter`

Topics

“Generate Parameter Samples for Sensitivity Analysis”

removeParameter

Class: `sdo.ParameterSpace`
Package: `sdo`

Remove parameter from `sdo.ParameterSpace` object

Syntax

```
ps = removeParameter(ps0,p)
```

Description

`ps = removeParameter(ps0,p)` removes the parameter, `p`, from the `sdo.ParameterSpace` object, `ps0`, and returns the updated object, `ps`.

Input Arguments

`ps0`

Parameter space, specified as an `sdo.ParameterSpace` object.

`p`

Parameters to be removed, specified as one of the following:

- Vector of `param.Continuous` objects — Parameter objects. For example, `p = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac')`.
- Parameter name, specified as a character vector or string. For example, `'Ac'`.

Output Arguments

`ps`

Updated parameter space, returned as an `sdo.ParameterSpace` object.

Examples

Remove Parameter From `sdo.ParameterSpace` Object

Create an `sdo.ParameterSpace` object, `ps`, for the `Ac` and `K` parameters of the `sdoHydraulicCylinder` model.

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

```
ps =
  ParameterSpace with properties:
```

```
            ParameterNames: {'Ac'  'K'}
    ParameterDistributions: [1x2 prob.UniformDistribution]
           RankCorrelation: []
                   Options: [1x1 sdo.SampleOptions]
                     Notes: []
```

Remove K from ps.

```
ps = removeParameter(ps,p(2))
```

```
ps =
  ParameterSpace with properties:

            ParameterNames: {'Ac'}
    ParameterDistributions: [1x1 prob.UniformDistribution]
           RankCorrelation: []
                   Options: [1x1 sdo.SampleOptions]
                     Notes: []
```

Remove Ac from ps using the parameter name.

```
ps = removeParameter(ps,'Ac');
```

See Also

sdo.ParameterSpace | addParameter | sdo.getParameterFromModel

setDistribution

Class: `sdo.ParameterSpace`
Package: `sdo`

Set distribution of parameter in `sdo.ParameterSpace` object

Syntax

```
ps = setDistribution(ps0,p,pdist)
```

Description

`ps = setDistribution(ps0,p,pdist)` updates the `ParameterDistributions` property of the `sdo.ParameterSpace` object, `ps0`, for the specified parameters, `p`, and returns the updated object, `ps`.

Input Arguments

`ps0`

Parameter space, specified as an `sdo.ParameterSpace` object.

`p`

Parameters whose distributions are to be updated, specified as one of the following:

- Vector of `param.Continuous` objects — Parameter objects. For example, `p = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac')`.
- Parameter name, specified as a character vector or string. For example, `'Ac'`.

`pdist`

Probability distribution for 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

Set Distribution of Parameters in Parameter Space

Create an `sdo.ParameterSpace` object for the `Ac` and `K` parameters of `sdoHydraulicCylinder` model.

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

By default, a uniform distribution is specified for all parameters in `p`.

```
ps.ParameterDistributions
```

```
ans =
  1x2
```

Specify a 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`

sdo.setCheckBlockEnabled

Package: sdo

Enable or disable all check blocks in model

Syntax

```
chk_blk_state = sdo.setCheckBlockEnabled(modelname,state)
```

Description

`chk_blk_state = sdo.setCheckBlockEnabled(modelname,state)` 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.

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.

Input Arguments

`modelname`

Simulink model name, specified as a character vector or string. For example, `'pidtune_demo'`.

The model must be open.

`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 `sdo.setCheckBlockEnabled`.

Output Arguments

`chk_blk_state`

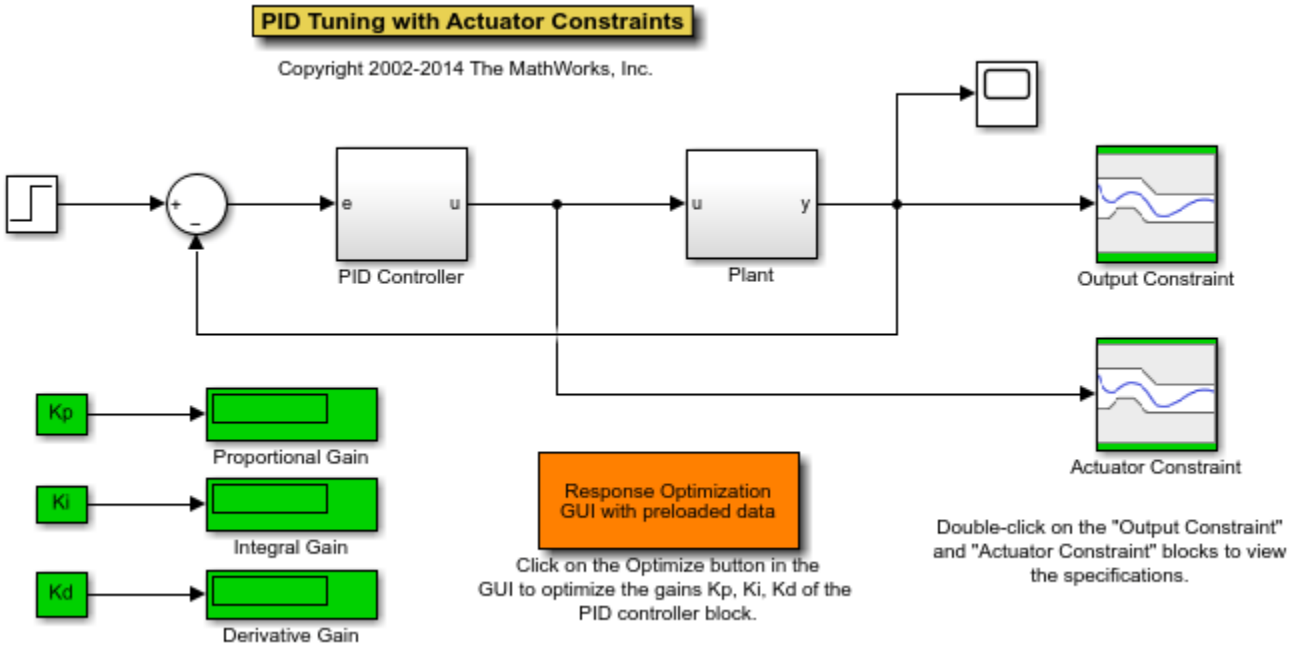
Original values of the `Enabled` block parameter of the model check blocks, returned as a cell array of character vectors.

Examples

Disable Model Check Blocks

Open the model.

```
modelname = 'pidtune_demo';
open_system(modelname);
```



Disable the model check blocks in a model.

```
state = 'off';
chkBlkState = sdo.setCheckBlockEnabled(modelname,state);
```

Restore the enabled state of the model check blocks.

```
sdo.setCheckBlockEnabled(modelname,chkBlkState);
```

Alternatives

You can open each model verification block in a model and select or clear the **Enable assertion** check box.

See Also

Introduced in R2012b

sdo.setValueInModel

Package: sdo

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, specified as a character vector or string. For example, `'sldo_model1'`.

`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`. Specify `param_des` as a `param.Continuous` object if you have a variable in a referenced model.
- Character vector or string for one variable. For multiple variables, specify as cell array of character vectors or a string array. For example, `{'Kp','Ki'}`.

  Also specify the `value` argument.

If a parameter is in a referenced model, the variable name must include the path. For instance, if a parameter `Ki` is in a referenced model named `Controller` used in a top-level model, use `sdo.setValueInModel('TopLevelModel','Controller:Ki',value)`.

If `Ki` is a model argument in a referenced model, provide block path from top-level model as follows, `sdo.setValueInModel('TopLevelModel','TopLevelModel/ControlBlock:Ki',value)`. Here, `ControlBlock` is the block name in the referenced model.

`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. `value` is required if `param_des` is a character vector or string.

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`

Topics

“Design Optimization to Meet Step Response Requirements (Code)”
“Design Optimization to Track Reference Signal (GUI)”
“Design Optimization Using Frequency-Domain Check Blocks (GUI)”
“Design Optimization Tuning Parameters in Referenced Models (Code)”

Introduced in R2011b

sdo.optimize

Solve design optimization problem

Syntax

```
[optimParam,optimInfo] = sdo.optimize(optimFcn,param)
[optimParam,optimInfo] = sdo.optimize(optimFcn,param,options)
[optimParam,optimInfo] = sdo.optimize(prob)
```

Description

Solve an optimization problem to find the values of design variables that satisfy the design requirements.

Use `sdo.optimize` to solve a design optimization problem of the following form:

$$\min_p F(p) \text{ subject to } \begin{cases} C_{leq}(p) \le 0 \\ C_{eq}(p) = 0 \\ A \times p \le B \\ A_{eq} \times p = B_{eq} \\ lb \le p \le ub \end{cases}$$

Here:

- F is the cost (objective).
- p is a scalar or vector of design variables.
- C_{leq} and C_{eq} are the nonlinear inequality and equality constraints, respectively.
- A and B are the linear inequality constraints.
- A_{eq} and B_{eq} are the linear equality constraints.
- lb and ub are the lower and upper bounds on p, respectively.

`[optimParam,optimInfo] = sdo.optimize(optimFcn,param)` uses `fmincon`, the default optimization method, to find the parameter values `optimParam` that satisfy the requirements specified in `optimFcn`.

`[optimParam,optimInfo] = sdo.optimize(optimFcn,param,options)` solves the optimization problem with the optimization options specified in `options`. Use `sdo.OptimizeOptions` to set these options.

`[optimParam,optimInfo] = sdo.optimize(prob)` solves the optimization problem specified in a structure `prob` that contains the function to be minimized, design variables, and optimization options.

Examples

Optimize Model Response

Solve an optimization problem specified in `sdoExampleCostFunction`. The example cost function is included with the Simulink® Design Optimization™ software.

This example function computes the following cost:

$$f(x) = x^2$$

The function also includes the following nonlinear and linear inequality constraints, respectively:

$$x^2 - 4x + 1 \le 0$$

$$\frac{2x}{3} - 3 \le 0$$

For optimization, $f(x)$ is minimized, subject to satisfying the constraints. The example function also computes cost and constraint gradients.

Create a design variable, using an initial value of 1.

```
p = param.Continuous('x',1);
```

Specify default optimization options to use `fmincon` solver. For this example, to include cost and constraint gradient evaluation from `sdoExampleCostFunction`, set the `GradFcn` option to `'on'`.

```
opts = sdo.OptimizeOptions;
opts.GradFcn = 'on';
```

Optimize the parameter.

```
[pOptim,optim_info] = sdo.optimize(@(p) sdoExampleCostFunction(p),p,opts);
```

```
 Optimization started 27-Feb-2022 09:30:35

                              max                     First-order
 Iter F-count        f(x)   constraint    Step-size    optimality
    0      3           1            0
    1      5        0.09            0          0.7         0.59
    2      6   0.0716349     0.001047       0.0324       0.0129
    3      7   0.0717968    9.127e-08     0.000302     2.37e-06
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.
```

The optimized parameter value is stored in the `Value` property of the returned `param.Continuous` object `pOptim`.

```
pOptim.Value
```

```
ans = 0.2679
```

Input Arguments

`optimFcn` — Cost function to minimize

function handle

Cost function to minimize, specified as a function handle. 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:

Field	Value
`F`	Cost (objective) evaluated at `p`. The solver minimizes `F`. `F` is a scalar double.
`Cleq`	Nonlinear inequality constraint violations evaluated at `p`. The solver satisfies $Cleq(p) \le 0$. *Cleq* is a double *m*-by-1 vector, where *m* is the number of nonlinear inequality constraints.
`Ceq`	Nonlinear equality constraint violations evaluated at `p`. The solver satisfies $Ceq(p) = 0$. *Ceq* is a double *r*-by-1 vector, where *r* is the number of nonlinear equality constraints.
`leq`	Linear inequality constraint violations evaluated at `p`. The solver satisfies $leq(p) \le 0$. *leq* is a double *n*-by-1 vector, where *n* is the number of linear inequality constraints.
`eq`	Linear equality constraint violations evaluated at `p`. The solver satisfies $eq(p) = 0$. *eq* is a double *s*-by-1 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 `Cleq`, `Ceq`, `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:

Field	Value
`F`	Double *n*-by-1 vector that contains `dF(p)/dp`, where *n* is the number of scalar parameters
`Cleq`	Double *n*-by-*m* matrix that contains `dCleq(p)/dp`, where *m* is the number of nonlinear inequality constraints
`Ceq`	Double *n*-by-*r* matrix that contains `dCeq(p)/dp`, where *r* is the number of nonlinear equality constraints

You must return the derivatives of all applicable objective and constraint violations.

Example: For an example of a cost function that you can optimize using Simulink Design Optimization software, enter `edit sdoExampleCostFunction` in the MATLAB command window.

Data Types: `function_handle`

`param` — Model parameters to optimize
`param.Continuous` object | vector of `param.Continuous` objects

Model parameters to optimize, specified as a `param.Continuous` object or vector of `param.Continuous` objects.

- To specify a single parameter for optimization, use a `param.Continuous` object.
- To specify multiple parameters for optimization, use a vector of `param.Continuous` objects.

`options` — Optimization options
`sdo.OptimizeOptions` object

Optimization options, specified as an object created using `sdo.OptimizeOptions`. Use this option set to specify:

- Optimization method
- Maximum number of iterations
- Tolerances

`prob` — Optimization problem structure
structure

Optimization problem structure, specified as a structure with the following fields:

Field Name	Value
`OptFcn`	Cost function to minimize. See `optimFcn` for the input and output argument requirements of this function.
`Parameters`	Model parameters to optimize, specified as `param.Continuous` objects.
`Options`	Optimization options, created with `sdo.OptimizeOptions`.

Data Types: `struct`

Output Arguments

`optimParam` — Optimized parameter values

`param.Continuous` object | vector of `param.Continuous` objects

Optimized parameter values, returned as a `param.Continuous` object or vector of `param.Continuous` objects. The size of `optimParam` is the same as the size of `param`. The optimized parameter values are stored in the `Value` property of the returned `param.Continuous` object.

`optimInfo` — Optimization information

structure

Optimization information, returned as a structure with one or more of the following fields:

Field	Value
`F`	Optimized cost (objective) value.
`Cleq`	Optimized nonlinear inequality constraint violations. The field appears if you specify a nonlinear inequality constraint in `optimFcn`. The value is a *m*-by-1 vector, where the order of the elements corresponds to the order specified in `optimFcn`. Positive values indicate that the constraint is not satisfied. Check the `exitflag` field to confirm that the optimization succeeded.
`Ceq`	Optimized nonlinear equality constraint violations. The field appears if you specify a nonlinear equality constraint in `optimFcn`. The value is a double *r*-by-1 vector, where the order of the elements corresponds to the order specified in `optimFcn`. Any nonzero values indicate that the constraint is not satisfied. Check the `exitflag` field to confirm that the optimization succeeded.
`leq`	Optimized linear inequality constraint violations. The field appears if you specify a linear inequality constraint in `optimFcn`. The value is a double *n*-by-1 vector, where the order of the elements corresponds to the order specified in `optimFcn`. Nonzero values indicate that the constraint is not satisfied. Check the `exitflag` field to confirm that the optimization succeeded.

Field	Value
`eq`	Optimized linear equality constraint violations. The field appears if you specify linear equality constraints in `optimFcn`. The value is a double *s*-by-1 vector, where the order of the elements corresponds to the order specified in `optimFcn`. Nonzero values indicate that the constraint is not satisfied. Check the `exitflag` field 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 `optimFcn`.
`Jacobian`	Jacobian information at the optimized parameter values. The field appears if the solver specified in the `Method` property of `sdo.OptimizeOptions` computes Jacobians. The value is a structure whose fields are dependent on `optimFcn`.
`exitflag`	Integer identifying the reason the algorithm terminated. See `fmincon`, `patternsearch`, `fminsearch`, and `lsqnonlin` for a list of the values and the corresponding termination reasons.
`iterations`	Number of optimization iterations.
`SolverOutput`	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`, `fminsearch`, and `lsqnonlin` for a list of solver outputs and their description.
`Stats`	Structure that contains statistics collected during optimization, such as start and end times, number of function evaluations, and restarts.

Data Types: `struct`

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” (Global Optimization Toolbox) 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.

Alternative Functionality

Apps

The **Response Optimizer** app provides a graphical interface to specify design requirements and optimize model parameters. For more information, see “Design Optimization to Meet a Custom Objective (GUI)” and “Design Optimization to Track Reference Signal (GUI)”.

The **Parameter Estimator** app provides a graphical interface to specify experiments and tune model parameters so that the model output matches the experiment data. For more information, see “Estimate Model Parameter Values (GUI)”.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `UseParallel` property of the `sdo.OptimizeOptions` object to `true`.

For more information, see “Use Parallel Computing for Response Optimization” and “Use Parallel Computing for Parameter Estimation”.

See Also

`sdo.OptimizeOptions` | `param.Continuous`

Topics

“Design Optimization to Meet a Custom Objective (Code)”
“Design Optimization to Meet Step Response Requirements (Code)”
“Design Optimization to Meet Frequency-Domain Requirements (Code)”
“Estimate Model Parameter Values (Code)”

Introduced in R2011a

sdo.sample

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.

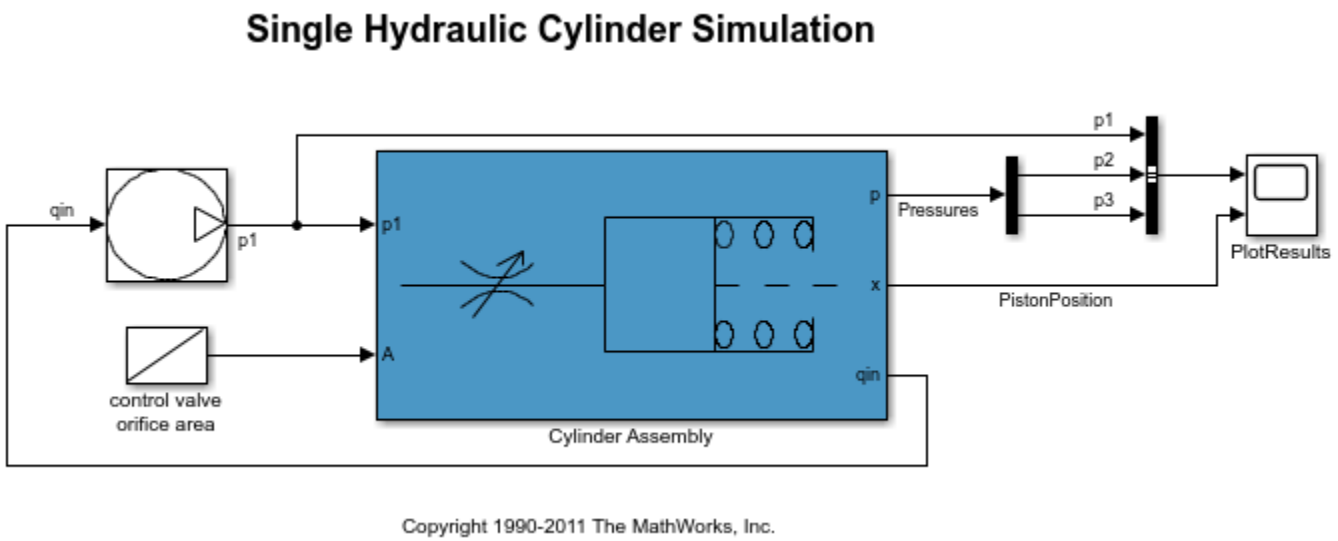
`x = sdo.sample(___,opt)` specifies sampling options such as the sampling method. This syntax can include any of the input argument combinations in the previous syntaxes.

Examples

Generate Parameter Samples

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);
```

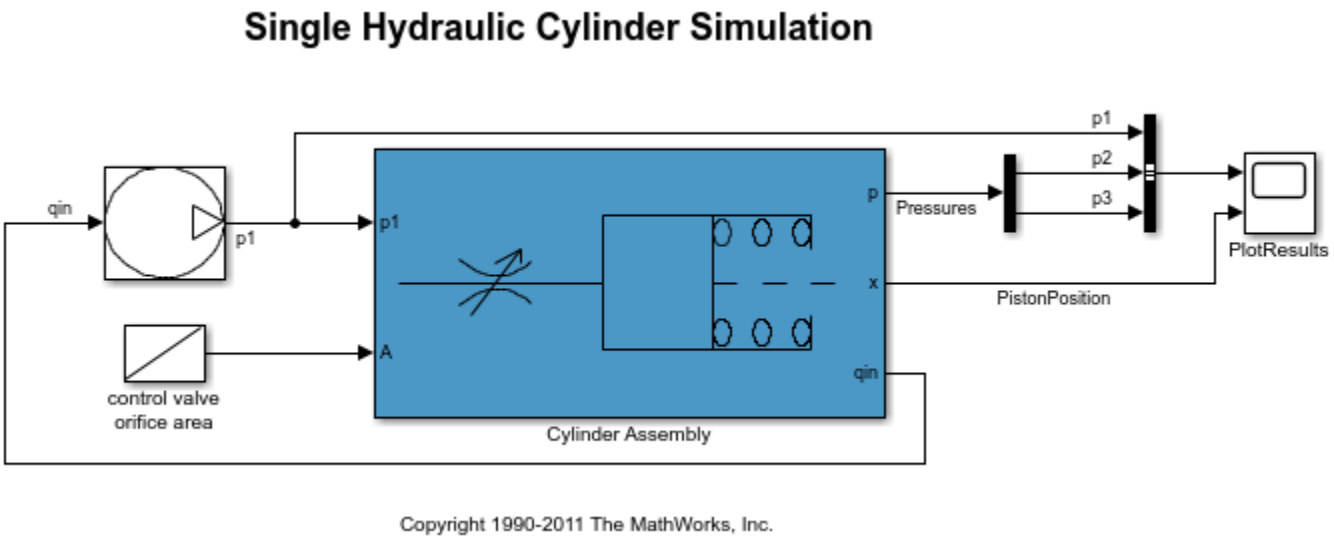
Generate samples for the parameters.

```
x = sdo.sample(ps);
```

Specify Number of Samples

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);
```

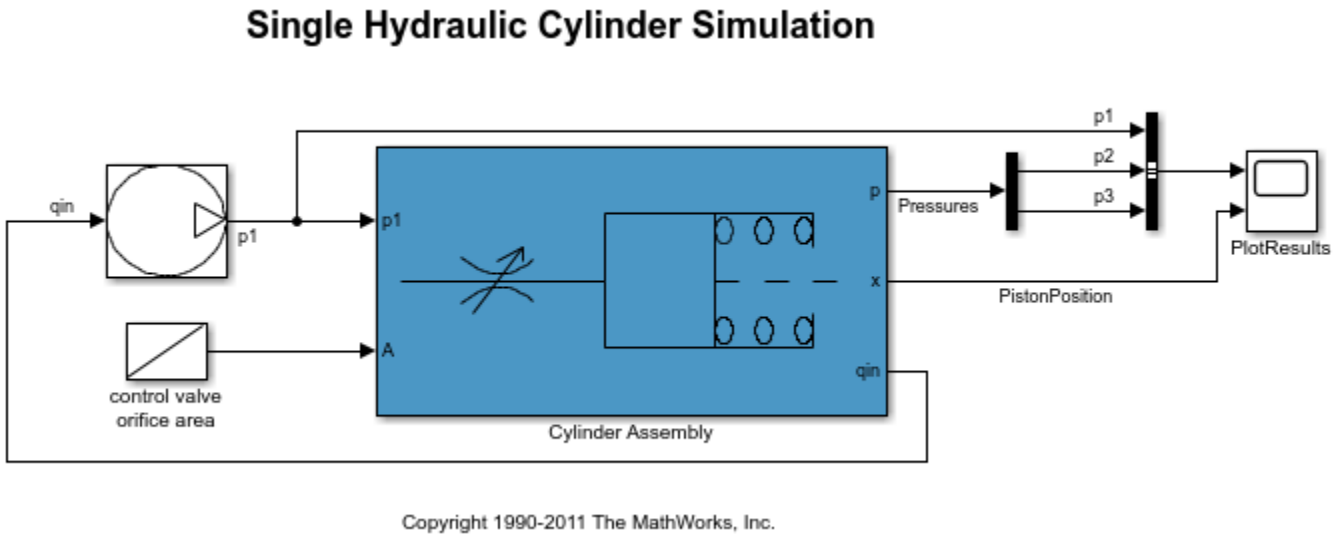
Generate 50 samples for the parameters.

```
x = sdo.sample(ps,50);
```

Specify Sampling Method

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 as Latin hypercube.

```
opt = sdo.SampleOptions;
opt.Method = 'lhs';
```

Generate 50 samples for the parameters using Latin hypercube sampling.

```
x = sdo.sample(ps,50,opt);
```

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 10*Np* 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 *Ns* 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`. If you specify `N`, *Ns* is equal to `N`. Otherwise, *Ns* is equal to 2*Np*+1.

See Also
`sdo.SampleOptions` | `sdo.evaluate`

Topics
“Design Exploration Using Parameter Sampling (Code)”
“Identify Key Parameters for Estimation (Code)”
“Generate Parameter Samples for Sensitivity Analysis”

Introduced in R2014a

fastRestart

Class: `sdo.SimulationTest`
Package: `sdo`

Simulate Simulink model in fast restart mode using simulation scenario

Syntax

```
Simulator_out = fastRestart(Simulator,EnablefastRestart)
```

Description

`Simulator_out = fastRestart(Simulator,EnablefastRestart)` configures a Simulink model and simulation scenario specified in `sdo.SimulationTest` object, `Simulator`, for simulation in fast restart mode. Fast restart configures the model to compile once when first simulated. Subsequent model simulations reuse the compiled data, speeding up subsequent simulation runs.

Input Arguments

`Simulator` — Simulation scenario for Simulink model

`sdo.SimulationTest` object (default)

Simulation scenario for Simulink model, specified as an `sdo.SimulationTest` object. A simulation scenario specifies input signals, model parameter and initial state values, and signals to log for a model.

If you want to linearize the model, for example if you have frequency-domain design requirements, specify the linearization logging information in the `SystemLoggingfInfo` property of `Simulator`.

`EnablefastRestart` — Configuration of model and simulation scenario for fast restart

`'on'` (default) | `'off'`

Configuration of model and simulation scenario for fast restart, specified as one of the following values:

- `'on'` — Initializes a Simulink model to simulate in the fast restart mode using simulation scenario specified in `Simulator` object. Fast restart configures the model to compile once when first simulated. Subsequent model simulations reuse the compiled data, speeding up subsequent simulation runs.

  Once you have initialized a model in fast restart, you can change only tunable properties of the model.
- `'off'` — Turns off the fast restart mode. Use this option to change nontunable properties of your model.

Output Arguments

`Simulator_out` — Simulation scenario configured for fast restart

`sdo.SimulationTest` object

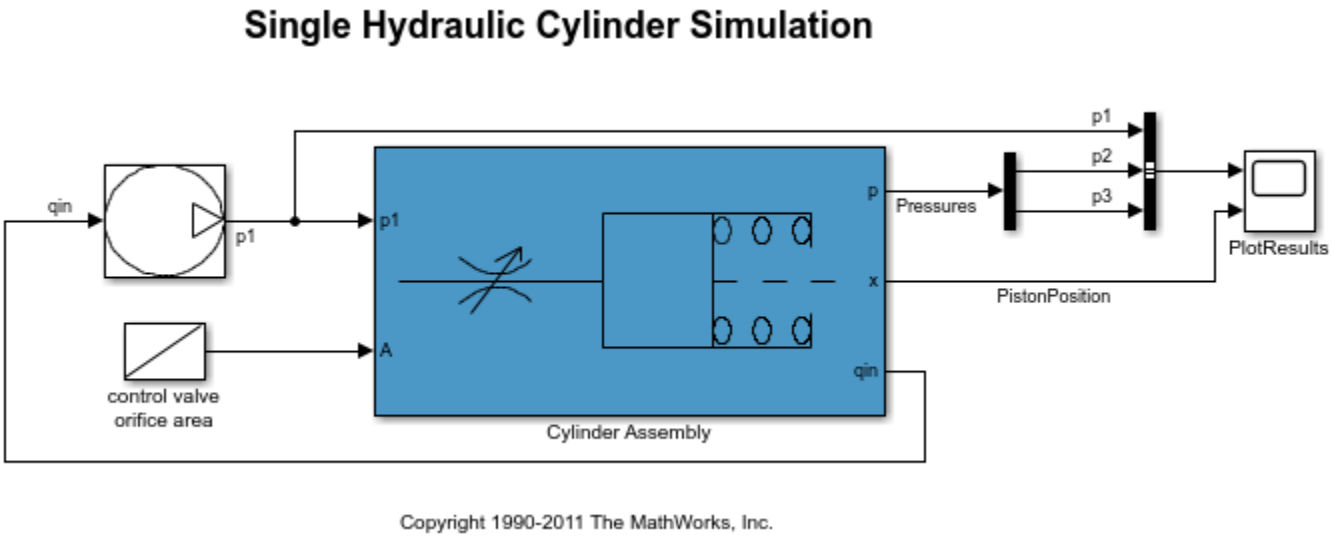
Simulation scenario configured for fast restart, returned as a `sdo.SimulationTest` object.

Examples

Simulate a Model in Fast Restart Mode

Open the model.

```
open_system('sdoHydraulicCylinder')
```



Create a simulation scenario for the model.

```
simulator = sdo.SimulationTest('sdoHydraulicCylinder');
```

Configure the model and `simulator` for fast restart.

```
simulator = fastRestart(simulator,'on');
```

Simulate the model.

```
simulator = sim(simulator);
```

The first simulation in fast restart mode requires the model to compile. Subsequent simulations execute in fast restart mode and reuse the compiled data.

Adjust tunable model parameters.

```
Ac = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac');
Ac.Value = 0.5;
simulator.Parameters = Ac;
```

Simulate the model again.

```
simulator = sim(simulator);
```

The model simulates in fast restart mode.

Turn off fast restart mode.

```
simulator = fastRestart(simulator,'off');
```

Tips

- To optimize, evaluate, or estimate a Simulink model, first create a simulator configured for fast restart (`Simulator_out`). Then use `Simulator_out` as an input to your cost function. If you create a simulator in the cost function, you cannot use fast restart mode.
- When you enable fast restart, you can change only tunable properties of the model.
- To linearize the model, specify the linearization logging information in the `SystemLoggingfInfo` property of `Simulator` before configuring the model for fast restart.

See Also

`sdo.optimize` | `sdo.evaluate` | `sdo.SimulationTest` | `sim` | `find` | `who`

Topics

“Improving Optimization Performance Using Fast Restart (Code)”
“Use Fast Restart Mode During Response Optimization”
“Use Fast Restart Mode During Sensitivity Analysis”

Introduced in R2015b

find

Class: `sdo.SimulationTest`
Package: `sdo`

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 character vector or string.

Example: `'sdoHydraulicCylinder'`

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` | `fastRestart` | `who`

Topics

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

prepareToDeploy

Class: `sdo.SimulationTest`
Package: `sdo`

Configure simulation scenario for deployment with Simulink Compiler

Syntax

```
Simulator_out = prepareToDeploy(Simulator)
Simulator_out = prepareToDeploy(Simulator,p)
```

Description

`Simulator_out = prepareToDeploy(Simulator)` configures the Simulink model and simulation scenario specified as the `sdo.SimulationTest` object `Simulator`, for deployment with Simulink Compiler.

`prepareToDeploy` saves logging and parameter information in the simulation scenario `Simulator_out` so that the `sdo.SimulationTest` object does not need to perform model error checking or configuration when deployed as part of an executable.

`Simulator_out = prepareToDeploy(Simulator,p)` configures the `sdo.SimulationTest` object `Simulator` for deployment with an optional vector `p` of `param.Continuous` objects that can be used to modify the model variables once the model is deployed.

Input Arguments

`Simulator` — Simulation scenario for Simulink model

`sdo.SimulationTest` object (default)

Simulation scenario for a Simulink model, specified as an `sdo.SimulationTest` object. A simulation scenario specifies input signals, model parameter and initial state values, and signals to log for a model.

`p` — Parameters to modify the deployed model

vector

Parameters to modify the deployed model, specified as a vector of `param.Continuous` objects. You can use `p` to modify the model variables once the model is deployed.

Output Arguments

`Simulator_out` — Simulation scenario configured for deployment with Simulink Compiler

`sdo.SimulationTest` object

Simulation scenario configured for deployment with Simulink Compiler, returned as an `sdo.SimulationTest` object.

Examples

Set Up Experiment and Simulation Test Objects for Deployment

For this example, set up your parameter estimation problem using the **Parameter Estimator** app and generate MATLAB code from it. For more information on generating MATLAB code from the app, see “Generate MATLAB Code for Parameter Estimation Problems (GUI)”. Alternatively, you can also set up your estimation problem at the command line.

Next, split the generated MATLAB code just before the estimation objective function is defined. This results in two files − a `run` function and a `setup` function, as described in “Deployed Application of Parameter Estimation”.

In the `setup` function, add the following lines of code at the end to configure the experiment and simulation test objects for deployment and save them to a MAT-file.

```
Experiment_out = prepareToDeploy(Experiement);
Simulator = createSimulator(Experiment_out);
Simulator = prepareToDeploy(Simulator,p);
save ObjectsToDeploy Experiment_out Simulator p
```

In the `run` function, add the following lines of code at the beginning of the function to include the Simulink model in the compiled code and load the objects that were saved in the `setup` function.

```
%#function simulink_model_name.slx
load ObjectsToDeploy Experiment_out Simulator p
```

Next, add the following lines to load the experiment data and update the experiment object. For this example, assume that the experiment data is contained in the first three columns of a Microsoft Excel spreadsheet file named `fname.xls`.

```
d = xlsread(fname);
u = timeseries(d(:,3),d(:,1));
y = timeseries(d(:,2),d(:,1));
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param1',u);
Experiment_out = updateIOData(Experiment_out,'simulink_model_name/param2',y);
```

For a detailed example showing how to deploy your parameter estimation problem using the Simulink Compiler, see “Deployed Application of Parameter Estimation”.

See Also

`sdo.SimulationTest` | `param.Continuous`

Topics

“Deployed Application of Parameter Estimation”

Introduced in R2020a

sim

Class: `sdo.SimulationTest`
Package: `sdo`

Simulate Simulink model using simulation scenario

Syntax

```
sim_out = sim(sim_obj)
sim_out = sim(sim_obj,Name,Value)
```

Description

`sim_out = sim(sim_obj)` simulates a Simulink model using the simulation scenario specified in `sim_obj`.

Before simulating the model, specify the parameter values and signals to log in the `Parameters` and `LoggingInfo` properties of `sim_obj`. The software restores the parameter values and logging settings to their original values after simulation.

`sim_out = sim(sim_obj,Name,Value)` specifies simulation parameters using one or more `Name,Value` pair arguments.

Input Arguments

`sim_obj` — Simulation scenario

`sdo.SimulationTest` object

Simulation scenario for a model, specified as an `sdo.SimulationTest` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `sim_out = sim(sim_obj,'TimeOut',20)` specifies the maximum simulation run time as 20 seconds.

You can specify any of the simulation parameters that the Simulink `sim` command accepts as `Name,Value` arguments. In addition, you can specify the following parameters.

`ErrorOnWarnings` — Treatment of warnings as errors

`false` (default) | `true`

Treatment of warnings as errors, specified as the comma-separated pair consisting of `'ErrorOnWarnings'` and `true` or `false`. When you specify `ErrorOnWarning` as `true`, the `sim` command treats warnings that are generated during simulation as errors.

Data Types: `logical`

`RestoreSettingsAfterSim` — Restoration of settings after simulation
`true` (default) | `false`

Restoration of settings after simulation, specified as the comma-separated pair consisting of `'RestoreSettingsAfterSim'` and one of `true` or `false`. When you specify `RestoreSettingsAfterSim` as `true`, the `sim` command restores model parameter and model signal logging changes after the simulation ends.

Data Types: `logical`

`OperatingPointSetup` — Operating point setup object
`sdo.OperatingPointSetup` object

Operating point setup object, specified as the comma-separated pair consisting of `'OperatingPointSetup'` and an `sdo.OperatingPointSetup` object.

`OperatingPointSetup` must be specified as an `sdo.OperatingPointSetup` object. If provided, the steady-state operating point is computed.

For more information about computing a steady-state operating point during model simulation, see `sdo.OperatingPointSetup`.

Data Types: `function_handle`

Output Arguments

`sim_out` — Simulation scenario object containing logged signal data
`sdo.SimulationTest` object

Simulation scenario object containing logged signal data, returned as an `sdo.SimulationTest` object. The logged data is stored in the `LoggedData` property of `sim_out`.

Examples

Store Logged Signal Data During Simulation

Simulate the `sdoHydraulicCylinder` model, and store the `Pressures` signal of the model.

Log the `Pressures` signal that is output from the first port in the Cylinder Assembly block of the model.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;
Pressures.BlockPath = 'sdoHydraulicCylinder/Cylinder Assembly';
Pressures.OutputPortIndex = 1;
```

Create a simulation scenario for the model, and specify the signal to log.

```
simulator = sdo.SimulationTest('sdoHydraulicCylinder');
simulator.LoggingInfo.Signals = [Pressures];
```

Specify parameter values for simulation.

```
Ac = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac');
Ac.Value = 0.5;
simulator.Parameters = Ac;
```

Simulate the model.

```
sim_obj = sim(simulator);
```

The specified signal `Pressures` is logged during simulation in the `LoggedData` property of `sim_obj`. After simulation, the `sim` command restores model parameter and model signal logging changes. If you want to preserve your changes, specify the `RestoreSettingsAfterSim` argument as `false`.

```
sim_obj = sim(simulator,'RestoreSettingsAfterSim',false);
```

See Also

`sdo.optimize` | `fastRestart` | `find` | `who` | `sdo.OperatingPointSetup`

Topics

“Design Optimization to Meet Step Response Requirements (Code)”
“Design Optimization to Meet a Custom Objective (Code)”
“Set Model to Steady-State When Estimating Parameters (Code)”

who

Class: sdo.SimulationTest
Package: sdo

List logged data names

Syntax

```
names = who(sim_obj)
```

Description

`names = who(sim_obj)` 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

List logged data set names.

Log model signals.

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;
Pressures.BlockPath       = 'sdoHydraulicCylinder/Cylinder Assembly';
Pressures.OutputPortIndex = 1;
```

Store logged signal data.

```
simulator = sdo.SimulationTest('sdoHydraulicCylinder');
simulator.LoggingInfo.Signals = Pressures;
simulator = sim(simulator);
```

Find logged data sets.

```
names = who(simulator);
```

See Also

`sdo.optimize` | `find` | `sim` | `fastRestart`

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

sdoupdate

Update model containing Signal Constraint block

Syntax

```
sdoupdate(modelname)
sdoupdate(modelname,noprompt)
session = sdoupdate(modelname)
```

Description

`sdoupdate(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 **Response Optimizer**, after prompting you to update. The model must be open.

`sdoupdate(modelname,noprompt)` updates the response optimization project without prompting you.

`session = sdoupdate(modelname)` returns the **Response Optimizer** session.

Input Arguments

`modelname`

Simulink model name, specified as a character vector or string. For example, `'sldo_model1'`.

`noprompt`

Whether to prompt you about updating the response optimization project (`false`) or not (`true`).

Default: `false`

Output Arguments

`session`

Response Optimizer session name.

See Also

Response Optimizer

Introduced in R2011b

Sensitivity Analyzer

Explore design space and determine most influential model parameters

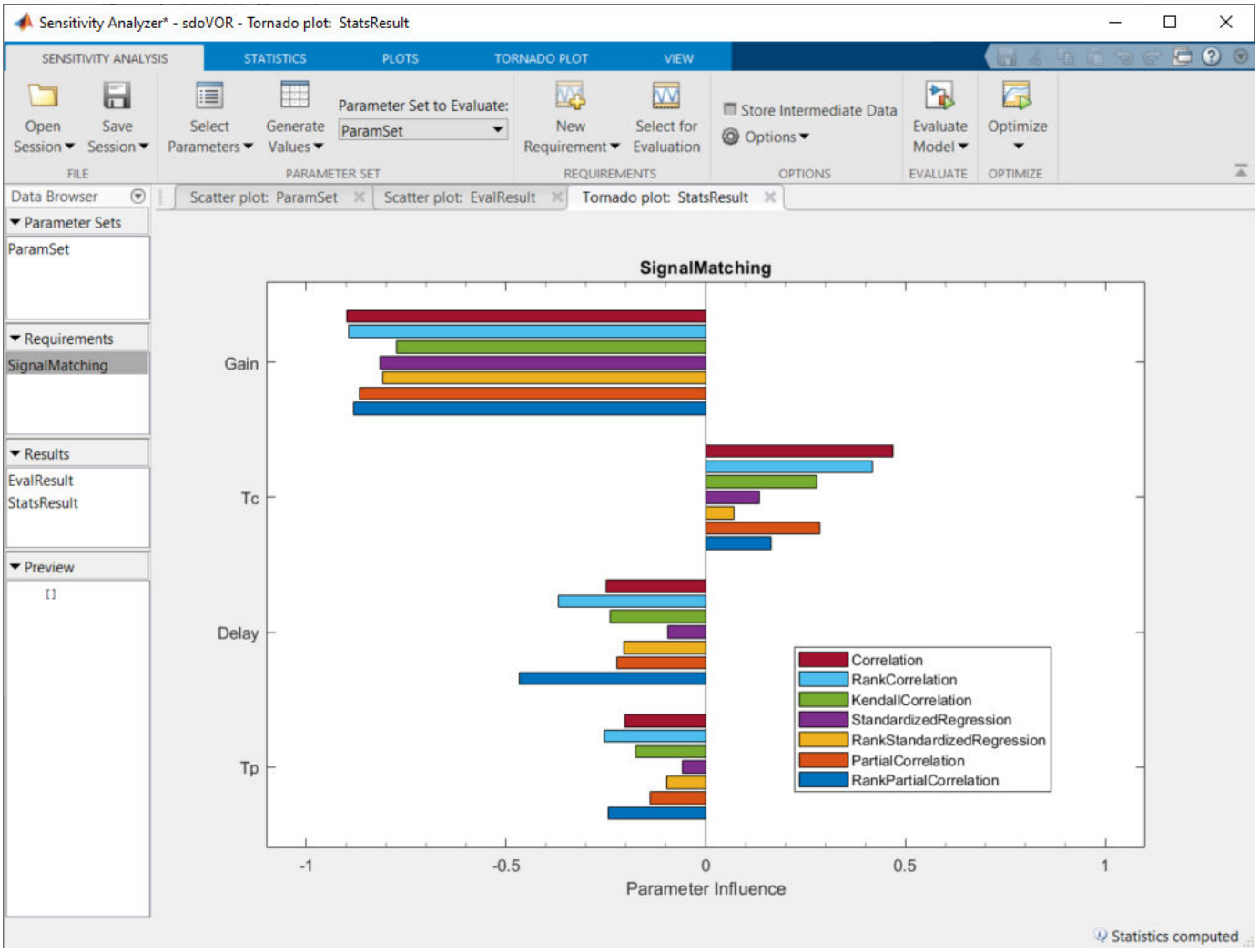
Description

The **Sensitivity Analyzer** app lets you explore the design space and determine the most influential Simulink model parameters using design of experiments, Monte Carlo simulations, and correlation analysis.

Using this app, you can:

- Select and sample parameters using design of experiments.
- Specify design requirements.
- Perform Monte Carlo simulations to evaluate the design requirement at selected parameter values.
- Analyze and visualize model sensitivity to parameters.

You can accelerate evaluation of design requirements using parallel computing and Simulink fast restart.



Open the Sensitivity Analyzer App

- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click the **Sensitivity Analyzer**.
- MATLAB command prompt: Enter `ssatool('modelname')`.

Examples

- “What is Sensitivity Analysis?”
- “Identify Key Parameters for Estimation (GUI)”
- “Design Exploration Using Parameter Sampling (GUI)”
- “Generate Parameter Samples for Sensitivity Analysis”
- “Evaluate Design Requirements”
- “Analyze Relation Between Parameters and Design Requirements”
- “Interact with Plots in the Sensitivity Analyzer”

Programmatic Use

`ssatool('modelname')` opens the **Sensitivity Analyzer** app and creates a new session. The model must be open or on the MATLAB path.

`ssatool(ssasession)` opens a previously saved **Sensitivity Analyzer** app session.

See Also

`sdo.evaluate`

Topics

“What is Sensitivity Analysis?”
“Identify Key Parameters for Estimation (GUI)”
“Design Exploration Using Parameter Sampling (GUI)”
“Generate Parameter Samples for Sensitivity Analysis”
“Evaluate Design Requirements”
“Analyze Relation Between Parameters and Design Requirements”
“Interact with Plots in the Sensitivity Analyzer”

Introduced in R2016a