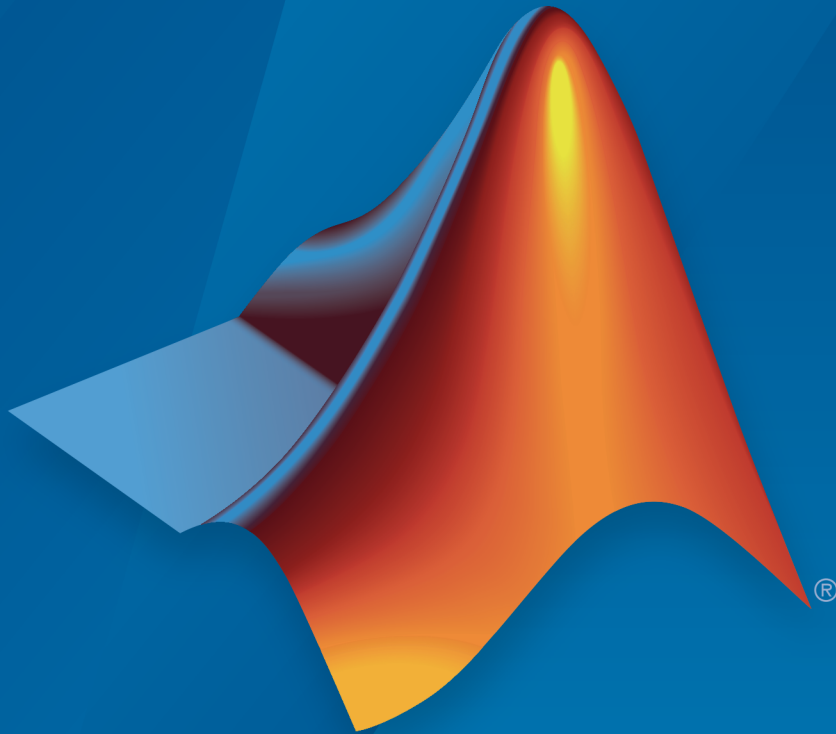# Simulink® Design Optimization™

User's Guide

MATLAB®&SIMULINK®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Revision History

March 2009	Online only	New for Version 1 (Release 2009a)
September 2009	Online only	Revised for Version 1.1 (Release 2009b)
March 2010	Online only	Revised for Version 1.1.1 (Release 2010a)
September 2010	Online only	Revised for Version 1.2 (Release 2010b)
April 2011	Online only	Revised for Version 1.2.1 (Release 2011a)
September 2011	Online only	Revised for Version 2.0 (Release 2011b)
March 2012	Online only	Revised for Version 2.1 (Release 2012a)
September 2012	Online only	Revised for Version 2.2 (Release 2012b)
March 2013	Online only	Revised for Version 2.3 (Release 2013a)
September 2013	Online only	Revised for Version 2.4 (Release 2013b)
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)

Contents

Data Analysis and Processing

1

2

Response Optimization

3

4

5

6

Data Analysis and Processing

- "Model Requirements for Importing Data" on page 1-2
- "Import Data" on page 1-6
- "Plot and Analyze Data" on page 1-13
- "Preprocessing Data" on page 1-16
- "Add Preprocessed Data Sets to Estimation Project (GUI)" on page 1-23
- "Export Prepared Data to the MATLAB Workspace" on page 1-26

Model Requirements for Importing Data

In this section...
"Select Input Signals" on page 1-3
"Select Output Signals" on page 1-4

Before you can analyze and preprocess the estimation data, you must assign the data to the model ports or signals. In order to assign the data, the Simulink® model must contain one of the following elements:

• Top-level Inport block

Note: You do not need an Inport block if your model already contains a fixed input block, such as a Step block.

• Top-level Outport block

• Logged signal, which can be a top-level signal in the model or a signal in a model subsystem

To enable signal logging for a signal, in the Simulink Editor, select the signal, click the **Record** button arrow ▾ and click **Log/Unlog Selected Signals**. For more information, see "Export Signal Data Using Signal Logging" in the Simulink documentation.

When you create an experiment, as described in "Create Experiment" on page 2-4, the top level input and output ports as well as logged signals are selected by default. You can add or remove the input and output signals using the experiment editor. In the experiment editor, the rows in the **Inputs** panel correspond to the model's top-level Inport blocks.
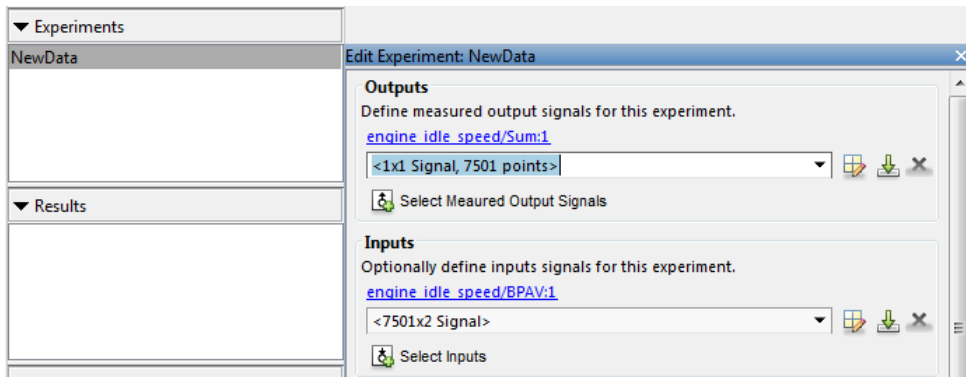
Similarly, the rows in the **Outputs** panel correspond to either the top-level Outport blocks or logged signals in the model.

Select Input Signals

You can add the Inport block in the experiment editor by clicking the **Select Inputs** button in the **Inputs** panel to launch the **Select Inputs** dialog box. You can select the Inport block you want by selecting the check box corresponding to it and clicking **OK**. There is only one Inport block for the engine_idle_speed model.
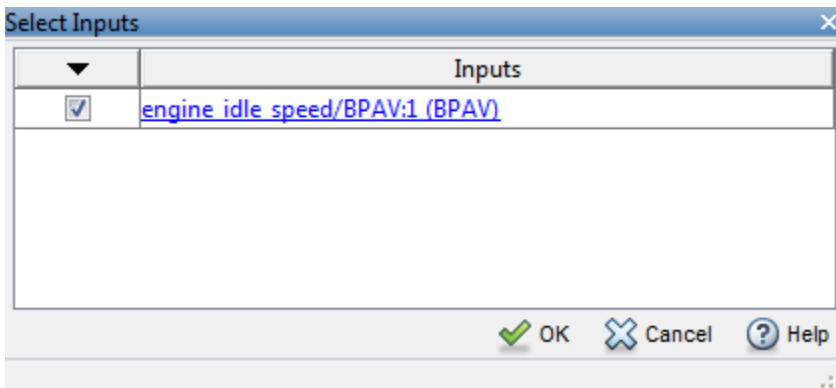
Using the dialog box, you can import the input data by typing, for example, [time,iodata(:,1)] in the **Inputs** panel. To learn more about importing data, see Import Data.
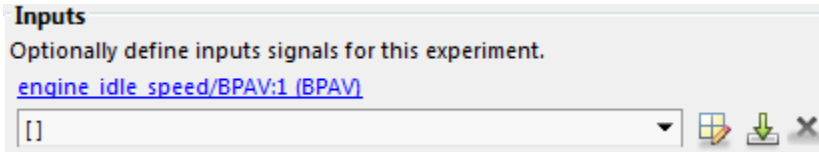
Select Output Signals

You can add the Output block in the experiment editor by clicking **Select Measured Output Signals** in the **Outputs** panel to launch the **Select Outputs** dialog box. You can select the Outport block you want by clicking the check box corresponding to it, and clicking **OK**. There is only one Outport block for the engine_idle_speed model.
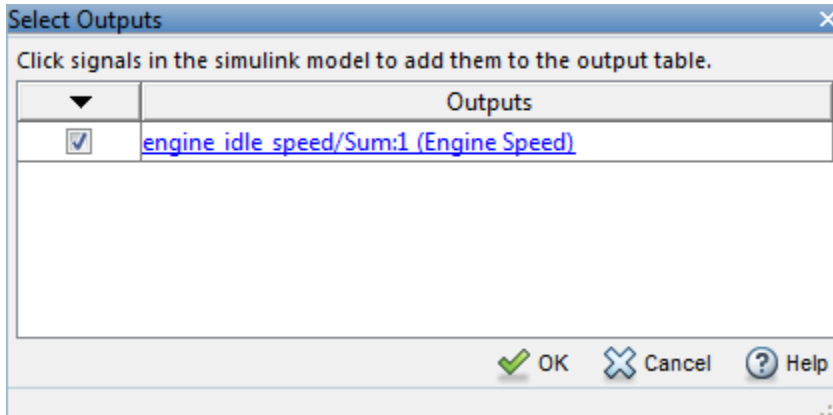
Using the dialog box, you can import the output data by typing, for example, [time,iodata(:,2)] in the **Inputs** panel. To learn more about importing data, see Import Data.
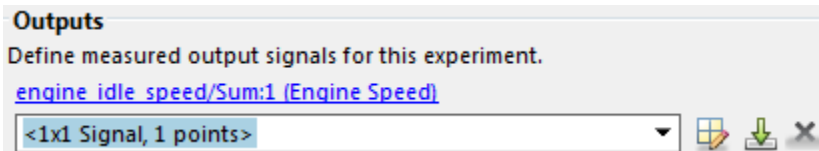
Related Examples

- "Import Data" on page 1-6

More About

- "What Is an Experiment?" on page 2-3

Import Data

In this section...
"Create Experiment" on page 1-6
"Time-Domain Data" on page 1-8
"Time-Series Data" on page 1-11
"Complex Data" on page 1-12

Create Experiment

Before you begin data import, create an experiment. Simulink Design Optimization™ software provides a tool for setting up the estimation session.

To create an estimation session:

1 At the MATLAB® prompt, open the nonlinear idle speed model of an automotive engine by typing :

```
engine_idle_speed
```
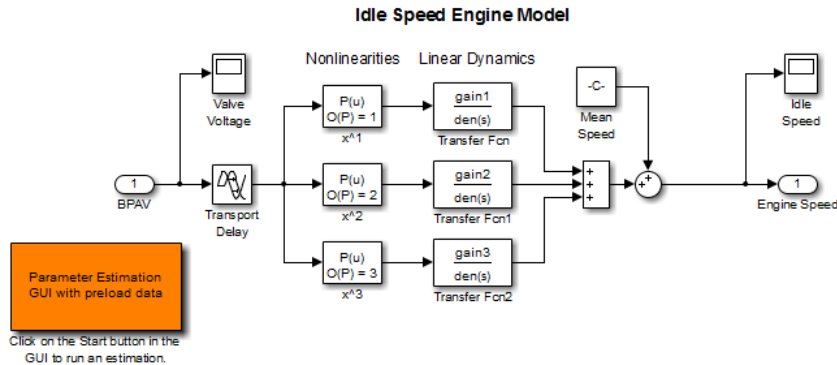

The model contains the Inport block BPAV and Outport block `Engine Speed` for importing input and output data, respectively. To learn more, see "Model Requirements for Importing Data" on page 1-2.

2 In the Simulink model window, open the Parameter Estimation tool by selecting **Analysis** > **Parameter Estimation**.
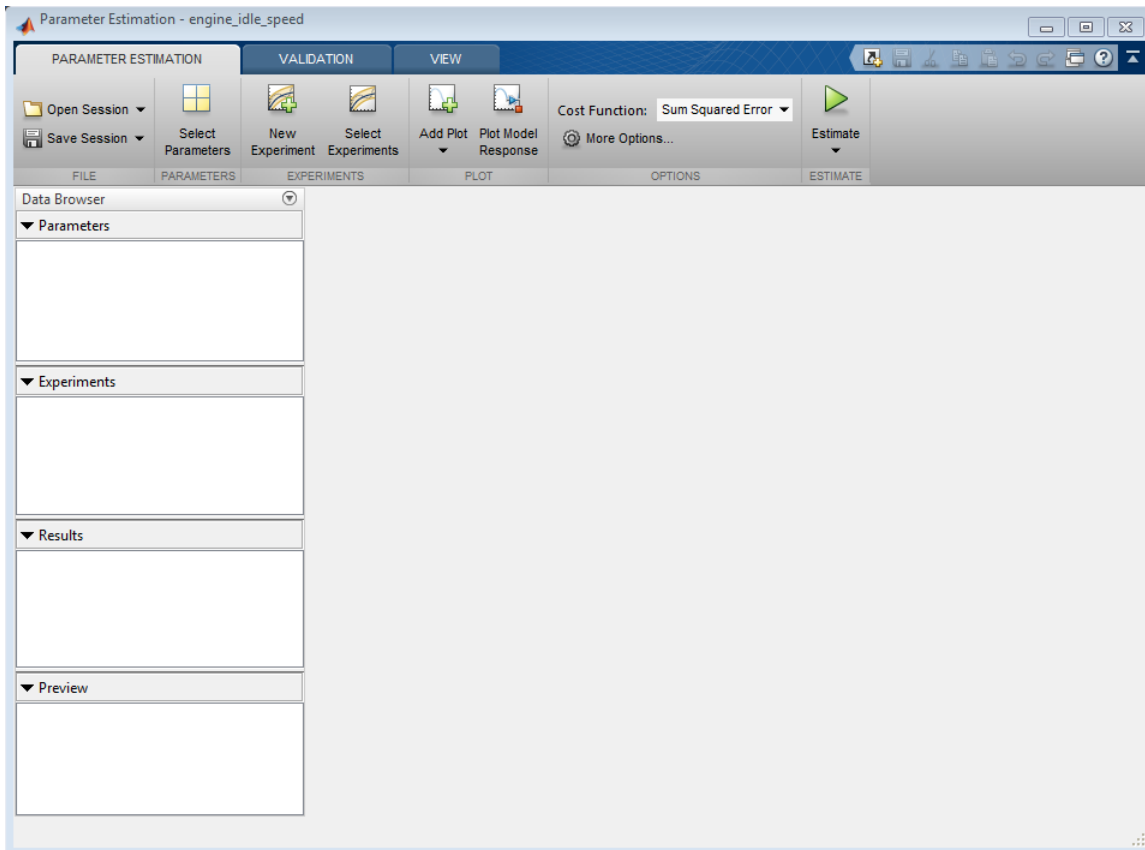
Parameter Estimation Tool

You can organize the estimation and validation tasks inside **Experiments** under **Data Browser** panel on the left. You can assign each experiment to an estimation task or validation task.

To create an experiment, click the **New Experiment** button.

1-7
1-7
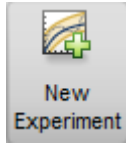
This creates an experiment called `Exp` under **Experiments**. To change the name of the experiment, right-click and select **Rename**. Call it `NewData`.

Note: The Simulink model must remain open to perform parameter estimation tasks.

Time-Domain Data

Experiments are collections of signal data, specifically input and output signal data. After you create an experiment, as described in "Create Experiment" on page 1-6, you can import data into your experiment from various sources including MATLAB® variables, MAT-files, Excel® files, or comma-separated-value files.

To import data into your experiment right-click and select **Edit...**. This will launch the experiment editor. In the experiment editor, you can define the signals contained in the experiment.
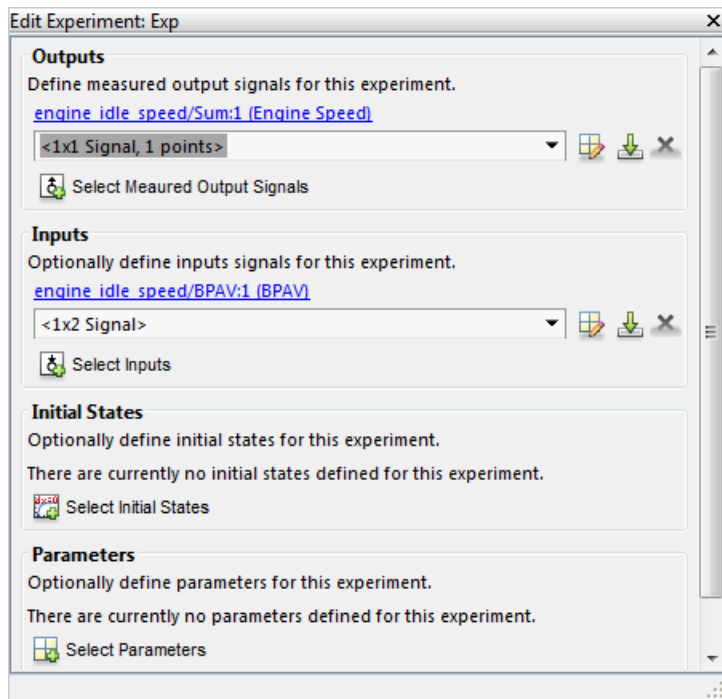
For example, the rows in the **Inputs** panel of the editor correspond to Inport block BPAV in the engine_idle_speed model.

The rows in the **Outputs** panel correspond to Outport block Engine Speed. You can import signal data from files or MATLAB workspace.

Note: The Simulink model must contain an Inport or Outport block or logged signals to enable importing data. For more information, see "Model Requirements for Importing Data". To select more output signals to specify data for, click **Select Measured Output Signals** in the **Outputs** panel.

The idle-speed model of an automotive engine contains the measured data stored in the iodata array in the workspace. The array contains two columns: the first for input data, and the second for output data. The time data is in the time array in the workspace. You can import the input data by typing [time,iodata(:,1)] in the **Inputs** panel.

Inputs

Optionally define inputs signals for this experiment.

engine idle speed/BPAV:1 (BPAV)

[time,iodata(:,1)]

Select Inputs
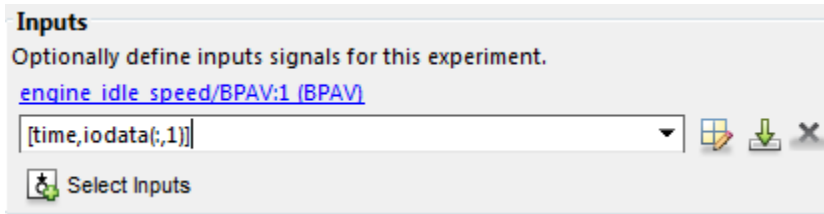
You can import the output data by typing [time,iodata(:,2)] in the **Outputs** panel.

You can view the data by clicking . The input data should look like this:

7501x2 double

	1	2	3	4
1	0	0		
2	0.0200	0		
3	0.0400	0		
4	0.0600	0		
5	0.0800	0		
	0.1000	0		

Output data should look like this:

7501x2 double

	1	2	3	4
1	0	713.2568		
2	0.0200	713.2568		
3	0.0400	709.0454		
4	0.0600	704.4067		
5	0.0800	699.8901		
	0.1000	699.6469		

After importing the data for `NewData` experiment, the experiment editor looks like this:
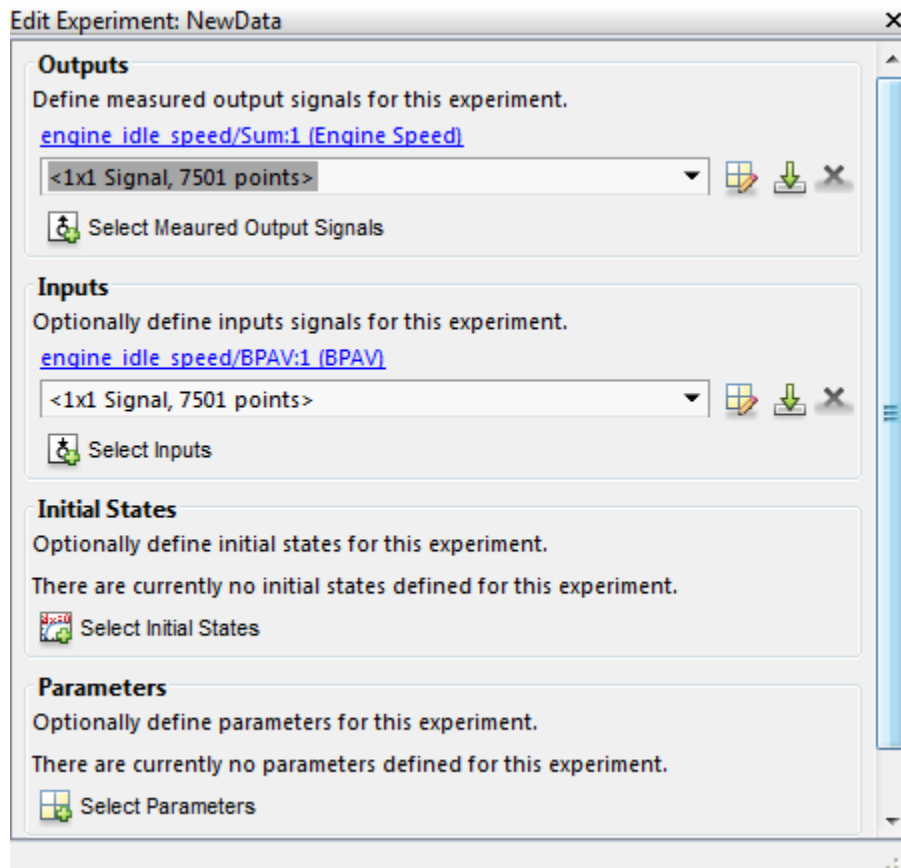
To import data from a file, click the ⬇ button.

To learn more about the **Edit Experiment:** dialog box, see "Experiment Editor" on page 2-22.

Time-Series Data

Time-series data is stored in time-series objects. For more information, see "Time Series Objects" in the MATLAB documentation.

When you import input data from a time-series object, `t`, for parameter estimation, you must specify the time vector and data as `[t.time,t.inputdata]` in the Inport signal dialog box. Similarly, to import output data, you must specify the time vector and data as `[t.time,t.outputdata]` in the Outport signal dialog box. For more information on how to import data into the experiment, see "Time-Domain Data" on page 1-8.

Complex Data

Complex-valued data is data whose value is a complex number. For example, a signal with the value `1+2j` is complex. You can use complex data to estimate parameters of electrical systems, such as the magnitude and phase.

Note: You must sample the real and imaginary parts of the data as a function of the same time vector.

To use complex data for parameter estimation:

1 Split the data into two data sets that contain the real and imaginary parts. To split the data, use the MATLAB functions `real`, and `imag`.

2 Create two signals, one for the real part and one for the imaginary part for the Inport or Outport block.

3 Select both signals in the experiment editor.

4 Import the data to the corresponding signal as described in "Time-Domain Data" on page 1-8.

Related Examples
- "Plot and Analyze Data" on page 1-13
- "Preprocessing Data" on page 1-16

More About
- "Model Requirements for Importing Data" on page 1-2

Plot and Analyze Data

In this section...
"Why Plot the Data Before Parameter Estimation" on page 1-13
"Plot Data" on page 1-13

Why Plot the Data Before Parameter Estimation

After you import the estimation data as described in "Import Data" on page 1-6, you can remove outliers, smooth, detrend, or otherwise treat the data to prepare for analysis and estimation. To view and analyze data characteristics, plot the data on a time plot.

Plot Data

Use an experiment plot to visualize experiment data. First, create an experiment and import data as described in "Import Data" on page 1-6. To create an experiment plot, in the Parameter Estimation tool, on the **Parameter Estimation** tab, click **Add Plot**, and select NewData under **Experiment Plots**.
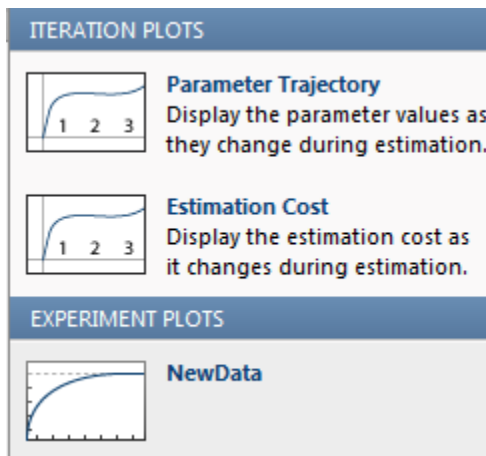
This creates plots of the input signal for the Inport block `BPAV` and output signal for the Outport block `Engine Speed` for the `engine_idle_speed` model (see "Create Experiment" on page 1-6).
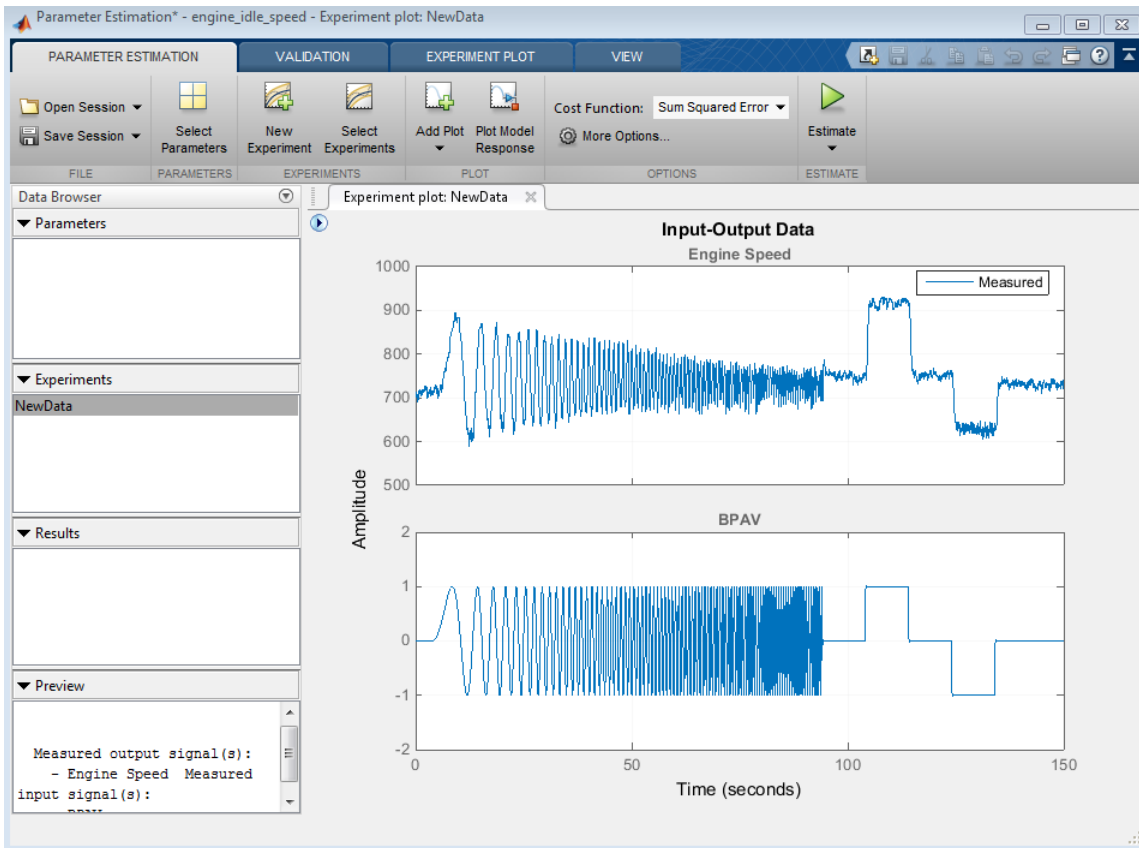
You can also plot the experiment data by right-clicking `NewData` and selecting **Plot measured experiment data** from the list.

Using the time plot, you can examine the data characteristics such as noise, outliers and portions of the data to use for estimating parameters. After you analyze the data, you can preprocess it as described in "Preprocessing Data" on page 1-16.

Related Examples

- "Import Data" on page 1-6
- "Preprocessing Data" on page 1-16

More About

- "Model Requirements for Importing Data" on page 1-2

Preprocessing Data

In this section...
"Ways to Preprocess Data" on page 1-16
"Remove Offset" on page 1-17
"Scale Data" on page 1-18
"Extract Data" on page 1-18
"Filter Data" on page 1-19
"Resample Data" on page 1-20
"Replace Data" on page 1-21

Ways to Preprocess Data

In the Parameter Estimation tool, you can preprocess imported data before you use it for parameter estimation. After plotting the measured data as shown in "Plot and Analyze Data"), you have access to the **Experiment Plot** tab.
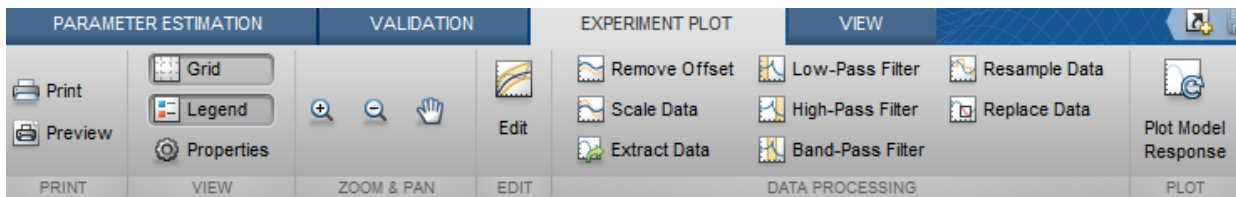
After importing the estimation data as described in "Import Data", you can perform the following preprocessing operations:

- "Remove Offset" on page 1-17 — Remove mean values, a constant value, or an initial value from the data.
- "Scale Data" on page 1-18 — Scale data by a constant value, signal maximum value, or signal initial value.
- "Extract Data" on page 1-18 — Exclude a portion of the data from the estimation process. You can select the data to exclude graphically or by specifying a start time and end time.
- "Filter Data" on page 1-19 — Smooth data using a low-pass, high-pass, or band-pass filter.

- "Resample Data" on page 1-20 — Resample data using zero-order hold or linear interpolation.
- "Replace Data" on page 1-21 — Replace data with a constant value, region initial value, region final value, or a line. You can use this functionality to replace outliers.

You can perform as many preprocessing operations on your data as are required for your application. For instance, you can both filter the data and remove an offset.

Remove Offset

On the **Experiment Plot** tab, click **Remove Offset**.
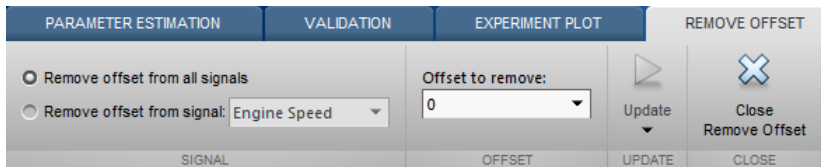
In the **Remove Offset** tab, you can choose to remove offset from all signals or specify the signal you choose. It is important for good identification results to remove data offsets. Select the value to remove from the **Offset to remove** drop down list. The options are:

- A constant value. Enter the value in the box. (Default: 0)
- Mean of the data, to create zero-mean data.
- Signal initial value.

As you change the offset value, the modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking

.

Or, to save the modified data values in a new experiment, click and select

.
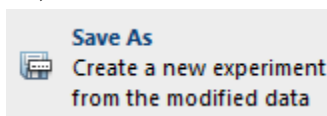
Scale Data

On the **Experiment Plot** tab, click **Scale Data**.
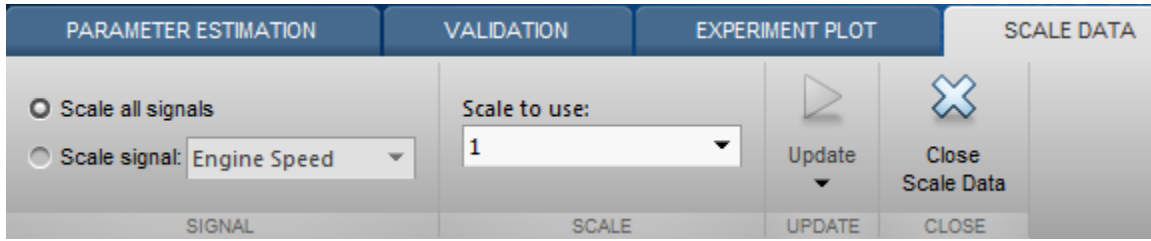
In the **Scale Data** tab, you can choose to scale all signals or specify a signal to scale. Select the scaling value from the **Scale to use** drop-down list. The options are:

- A constant value. Enter the value in the box. (Default: 1)
- Signal maximum value.
- Signal initial value.

As you change the scaling, the modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking

.

Or, to save the modified data values in a new experiment, click and select

.
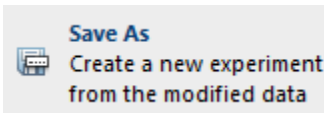
Extract Data

To extract a portion of your data to use in the estimation process, on the **Experiment Plot** tab, click **Extract Data**.
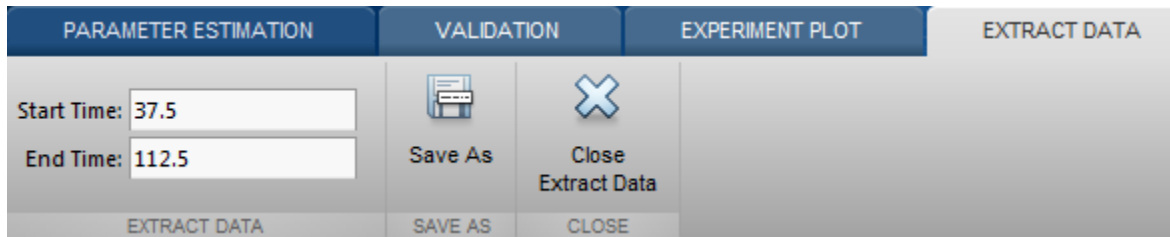
Select a subset of data to use for estimation in **Extract Data** tab. You can extract data graphically or by specifying start time and end time. To extract data graphically, click and drag the vertical bars to select a region of the data to use. Data outside the region that is highlighted yellow is discarded when you click .

After you choose the data to extract, you can save in a new experiment by clicking **Save As**.

Filter Data

You can filter your data using a low-pass, high-pass, or band-pass filter. A low-pass filter blocks high frequency signals, a high-pass filter blocks low frequency signals, and a band-pass filter combines the properties of both low- and high-pass filters. On the **Experiment Plot** tab click one of the **Low-Pass Filter**, **High-Pass Filter**, or **Band-Pass Filter** to open a new tab. For example, the low-pass filter tab appears as shown:
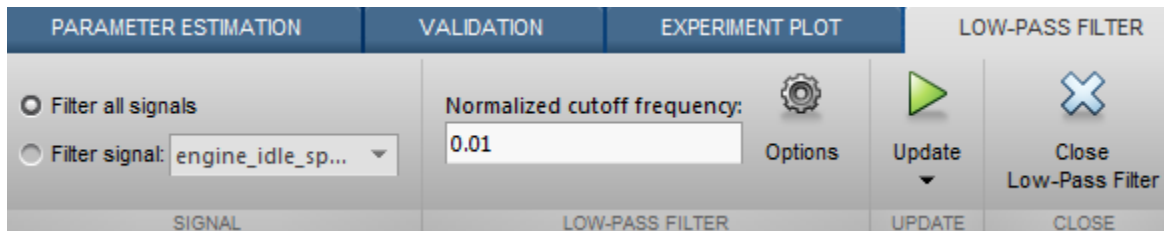
On the **Low-Pass Filter**, **High-Pass Filter**, or **Band-Pass Filter** tab, you can choose to filter all signals or specify a particular signal. For the low-pass and high-pass filtering, you can specify the normalized cutoff frequency of the signal. For the band-pass filter, you can specify the normalized start and end frequencies. Specify the frequencies by either entering the value in the associated field on the tab. Alternatively, you can specify

filter frequencies graphically, by dragging the vertical bars in the frequency-domain plot of your data.

Click **Options** to specify the filter order, and select zero-phase shift filter.

After making choices, update the existing data with the preprocessed data by clicking

.

Or, to save the modified data values in a new experiment, click and select

.
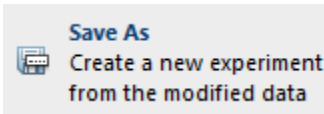
Resample Data

On the **Experiment Plot** tab, click the **Resample Data** button.
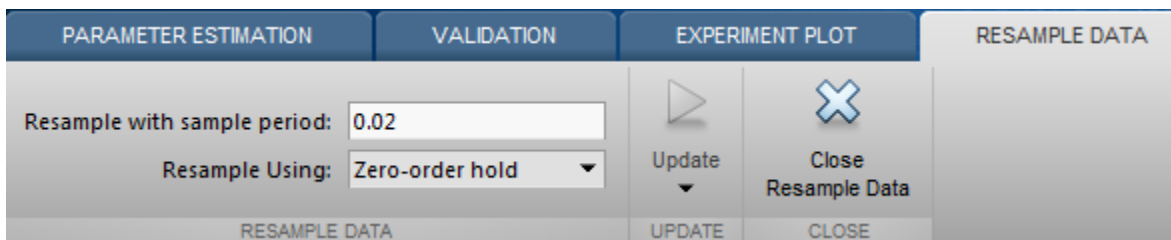
In the **Resample Data** tab, specify the sampling period using the **Resample with sample period:** field. You can resample your data using one of the following interpolation methods:

- `Zero-order hold` — Fill the missing data sample with the data value immediately preceding it.
- `Linear interpolation` — Fill the missing data using a line that connects the two data points.

By default, the resampling method is set to `zero-order hold`. You can select the `linear interpolation` method from the **Resample Using** drop-down list.

The modified data is shown in preview in the plot.

After making choices, update the existing data with the preprocessed data by clicking

.

Or, to save the modified data values in a new experiment, click and select

.

Replace Data

On the **Experiment Plot** tab click the **Replace Data** button.
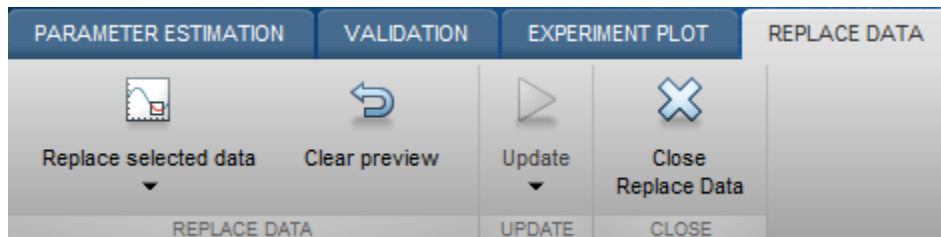
In the **Replace Data** tab, select data to replace by dragging across a region in the plot. Once you select data, choose how to replace it using the **Replace selected data** drop-down list. You can replace the data you select with one of these options:

- A constant value
- Region initial value
- Region final value
- A line

The replaced preview data changes color and the replacement data appears on the plot. At any time before updating, click **Clear preview** to clear the data you replaced and start over.

After making choices, update the existing data with the preprocessed data by clicking

.

Or, to save the modified data values in a new experiment, click and select

.
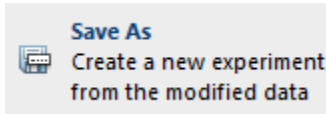
Replace Data can be useful, for example, to replace outliers. Outliers are data values that deviate from the mean by more than three standard deviations. When estimating parameters from data containing outliers, the results may not be accurate. Hence, you might choose to replace the outliers in the data before you estimate the parameters.

Related Examples

- "Import Data"

More About

- "Model Requirements for Importing Data"

Add Preprocessed Data Sets to Estimation Project (GUI)

After you preprocess the data using the techniques described in "Preprocessing Data" on page 1-16, you can add the data set to an estimation project either by overwriting an existing data set or creating a new data set.

In this section...
"Overwriting an Existing Data Set" on page 1-23
"Creating a New Data Set" on page 1-24

Overwriting an Existing Data Set

To overwrite an existing data set with the preprocessed data:

1   In the **Write results to** area of the Data Preprocessing Tool GUI, select the **existing dataset** option.

2   Choose the data set you want to overwrite from the drop-down list.
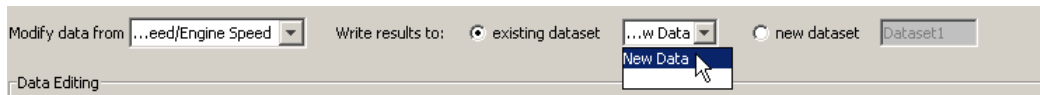
3   Click **Add**.

This action overwrites the selected data set with the modified data in the Control and Estimation Tools Manager GUI.
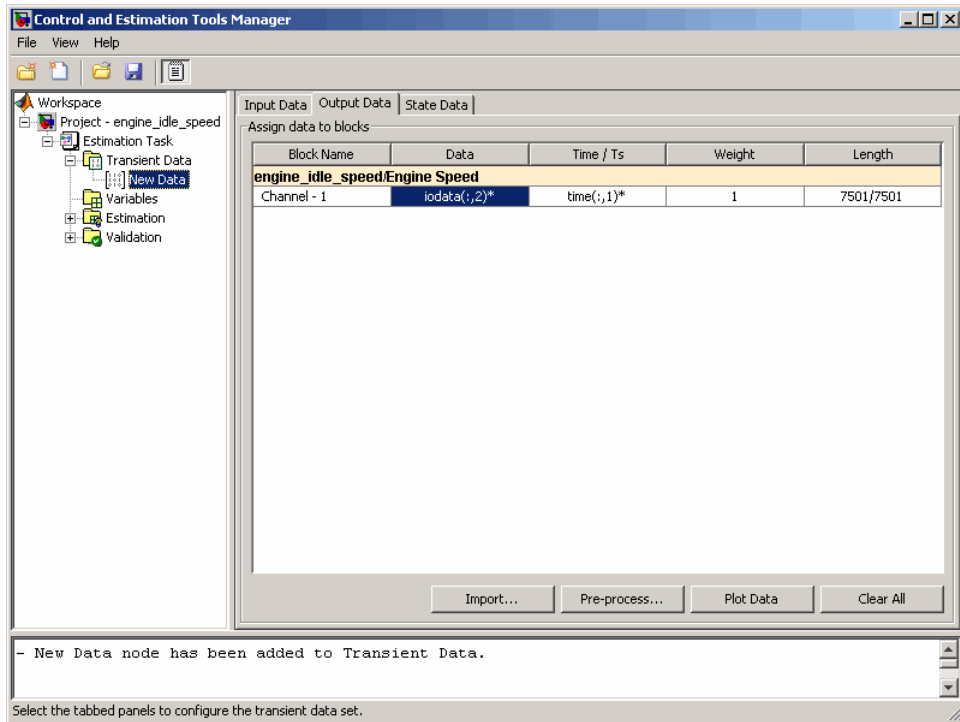
Tip You can export the preprocessed data to the MATLAB Workspace, as described in "Export Prepared Data to the MATLAB Workspace" on page 1-26.

Creating a New Data Set

If you do not want to overwrite an existing data set with the preprocessed data, as described in "Overwriting an Existing Data Set" on page 1-23, you can create a new data set for the preprocessed data:

1 In the **Write results to** area of the Data Preprocessing Tool GUI, select the **new dataset** option.

2 Specify the name of the data set in the adjacent field.

Modify data from [...eed/Engine Speed ▼] Write results to: ○ existing dataset [...w Data ▼] ● new dataset [Prep_Data]

3 Click **Add**.

This action adds a new data node in the Control and Estimation Tools Manager GUI containing the modified data.
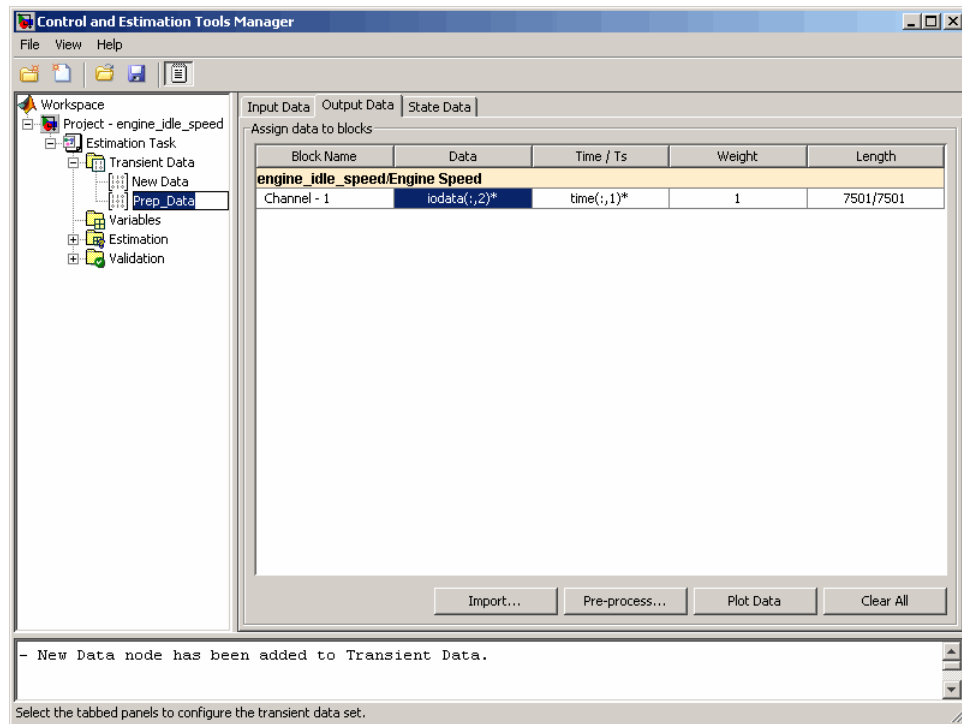
Tip You can export the preprocessed data to the MATLAB Workspace, as described in "Export Prepared Data to the MATLAB Workspace" on page 1-26.

Export Prepared Data to the MATLAB Workspace

After you add the preprocessed data to an estimation project, as described in "Add Preprocessed Data Sets to Estimation Project (GUI)" on page 1-23, you can export the data set to the MATLAB Workspace. You can use the data to further prepare it or estimate parameters using the data.

1 In the **Transient Data** node of the Control and Estimation Tools Manager GUI, select the node containing the prepared data set.

2 Right-click the table **Data** cell containing the data that you want to export, and select **Export**.

The Export to Workspace dialog box opens.

3 Specify the MATLAB variable names for the prepared data and the corresponding time vector in the **Data** and **Time** fields, respectively.
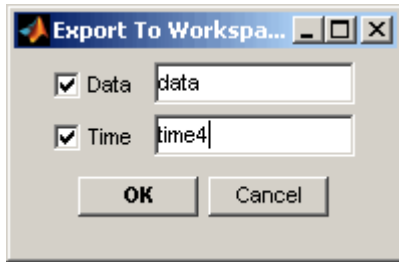
4 Click **OK**.

The resulting MATLAB variables `data` and `time4` appear in the MATLAB Workspace browser.

2

Parameter Estimation

What Is an Experiment?

To estimate unknown parameter values of a Simulink model, first create an experiment. An experiment specifies measured input and output data. During estimation, the experiment input data is used to simulate the model and the model output is compared with the measured experiment output data. For more information about creating experiments and importing data, see "Specify Estimation Data" on page 2-4.

In an experiment, you can specify initial-state values. 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 experiment. For more information, see "Specify Known Initial States" on page 2-18.

To estimate a model parameter on a per-experiment basis, specify the model parameter for each experiment. You can specify the initial values and limits for the parameter value for any of the experiments. Alternatively, you can specify a parameter value as a known quantity, not to be estimated. For more information, see "Specify Parameters for Estimation" on page 2-8. You can choose to update experiments with estimated model initial states and parameter values, or save the results in a new experiment. For more information, see "Estimation Options" on page 2-31.

To use experiments for validating the estimated parameter values, see "Model Validation" on page 2-46.

Related Examples

- "Specify Estimation Data" on page 2-4
- "Specify Parameters for Estimation" on page 2-8
- "Specify Known Initial States" on page 2-18
- "Specify Experiments for Estimation" on page 2-24

More About

- "Experiment Editor" on page 2-22

Specify Estimation Data

In this section...
"Create Experiment" on page 2-4
"Specify Data" on page 2-6

Create Experiment

Before you specify estimation data, create an experiment. At the MATLAB prompt, open the nonlinear idle speed model of an automotive engine by typing

```
engine_idle_speed
```
In the Simulink model window, open the Parameter Estimation tool by selecting
Analysis > **Parameter Estimation**.
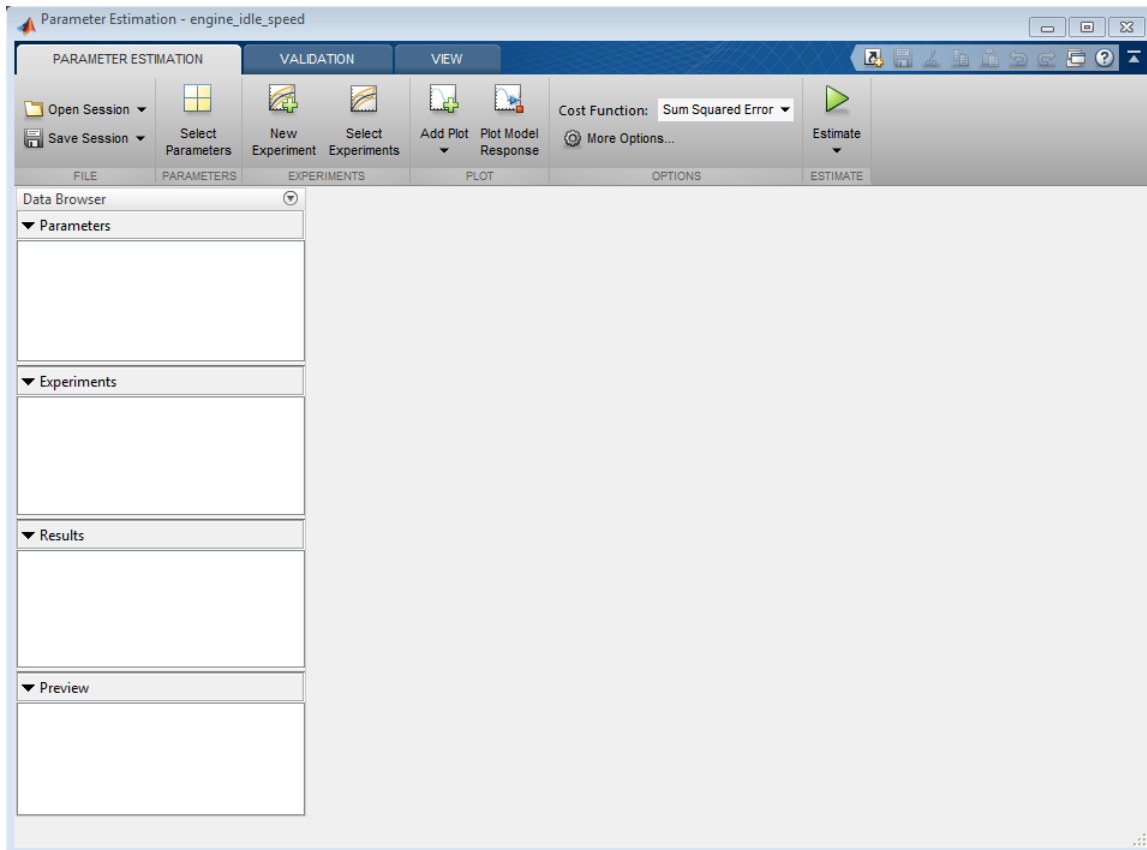
In the Parameter Estimation tool, on the **Parameter Estimation** tab, click **New Experiment**.
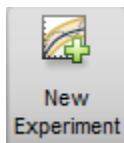
This action creates an experiment called `Exp` in the **Experiments** list in the **Data Browser** panel and opens the experiment editor. To change the name of the experiment, right-click `Exp` and select **Rename**. If you rename it `NewData`, the **Experiments** list now looks like this:
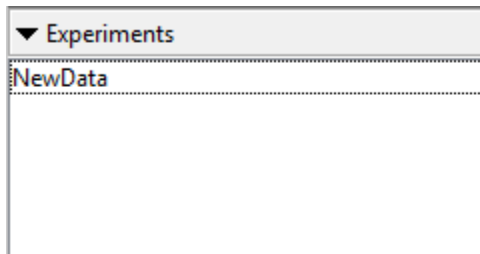
To learn more about how to further modify an experiment using the experiment editor, see "Experiment Editor" on page 2-22.

Specify Data

After you create the experiment, you can import the estimation data into the experiment. Launch the experiment editor by right-clicking on NewData and selecting **Edit...**.

The rows in the **Inputs** panel of the editor correspond to Inport block BPAV in the engine_idle_speed model. See "Import Data" on page 1-6.

The rows in the **Outputs** panel correspond to Outport block Engine Speed. You can import signal data from files or MATLAB workspace.

The idle-speed model of an automotive engine contains the measured data stored in the iodata array in the workspace. The array contains two columns: the first for input data, and the second for output data. The time data is in the time array in the workspace. You can import the input data by typing [time,iodata(:,1)] in the dialog box in the **Inputs** panel.
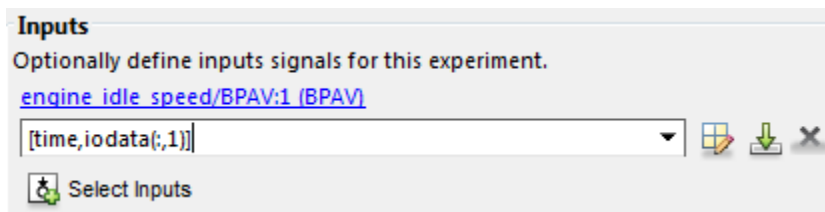
You can import the output data by typing [time,iodata(:,2)] in the dialog box in the **Outputs** panel.

> **Note:** You can have more than one input or output signal, but you can have only one data set for a signal. If you have multiple data sets, create multiple experiments.

Related Examples

- "Specify Parameters for Estimation" on page 2-8
- "Specify Independent Parameters to Estimate" on page 2-14
- "Specify Known Initial States" on page 2-18

More About

- "What Is an Experiment?" on page 2-3
- "Experiment Editor" on page 2-22

Specify Parameters for Estimation

In this section...
"Choosing Which Parameters to Estimate First" on page 2-8
"Specify Parameters for Estimation" on page 2-8

Choosing Which Parameters to Estimate First

Simulink Design Optimization software lets you estimate scalar, vector, and matrix parameters. You can take an iterative approach to estimating model parameters. For example, if you have a large number of parameters to estimate, start by estimating those that most influence the output. After you estimate a subset of parameters and validate the estimated parameters, you can select the remaining parameters for estimation.

Specify Parameters for Estimation

You can specify the parameters for estimation experiments using the **Estimated Parameters** editor. In the Parameter Estimation tool, on the **Parameter Estimation** tab, click **Select Parameters**.
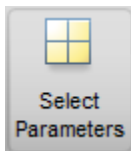
To select parameters for all experiments, click **Select Parameters** in the **Parameters Tuned for all Experiments** panel. This opens the **Select model variables** dialog. Here you can select the parameters you want to estimate by clicking the check box next to it or specifying an expression. For more information see "Select Parameters Using Select Model Variables Dialog Box" on page 2-11.

The editor looks like
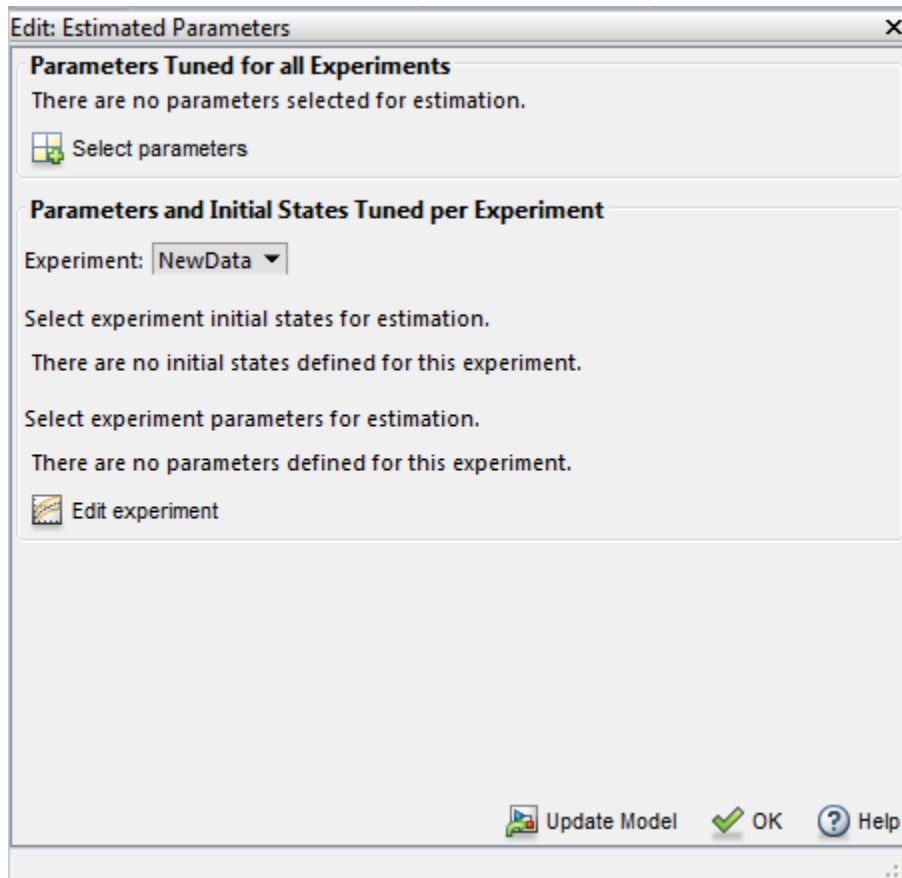
For this example, select `freq1`, `freq2`, `freq3`, `gain1`, `gain2`, `gain3` and `mean_speed` for estimation. You do not need to estimate the parameters all at once. You can first select all the parameters you are interested in, and then later select a subset to estimate. By default, all the parameters are selected for estimation. To deselect the ones you do not want to estimate, clear the **Estimate** check box for a parameter. For this example, only estimate `gain1`, `gain2`, `gain3` and `mean_speed`. Set their initial values 10, 100, 50, and 500, respectively, and then click **OK**. The **Edit: Estimated Parameters** dialog box looks like
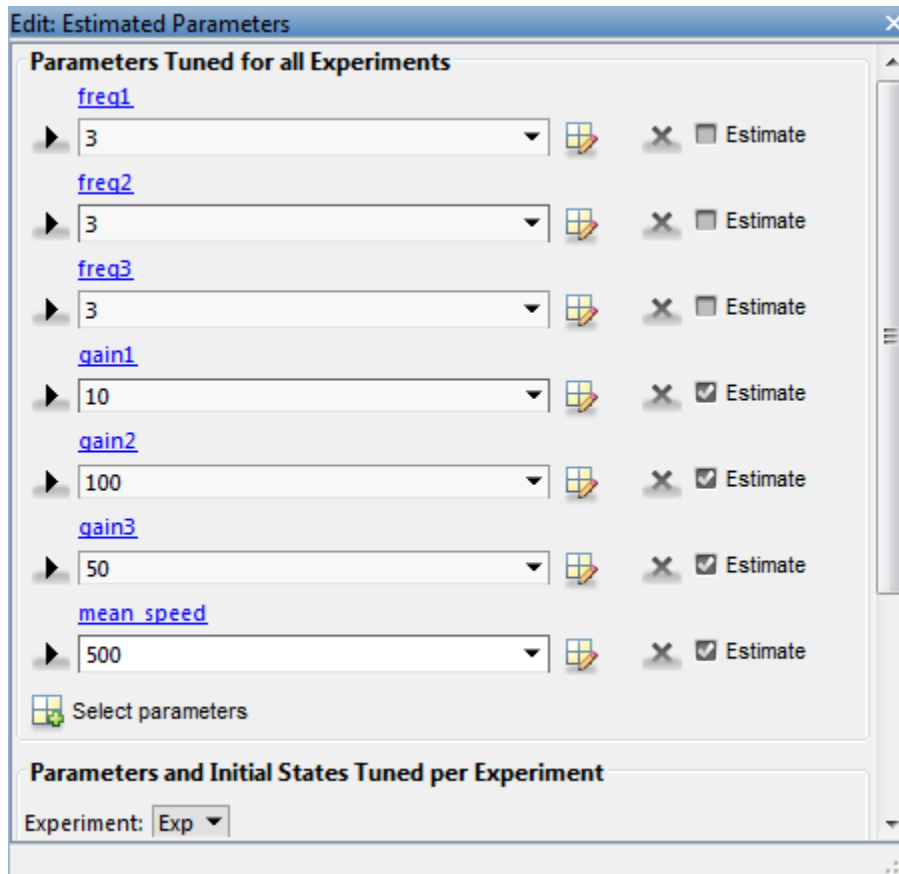
To learn how to specify initial values and upper and lower bounds of the parameters, see "Specifying Initial Guesses and Upper/Lower Bounds" on page 2-12.

Select Parameters to Estimate for a Specific Experiment

To select the parameters to estimate in a specific experiment, first, select the experiment for estimation as described in "Specify Experiments for Estimation" on page 2-24. Then, you can use the **Edit:Estimated Parameters** dialog to select parameters to estimate for that experiment. Select the experiment name from the **Experiment:** combo box in the **Parameters and Initial States Tuned per Experiment** panel. Then click **Edit experiment** to launch the experiment editor for the experiment you select.

Alternatively, you can right click the experiment name in the **Experiments** list and select **Edit...**. In the experiment editor, click the **Select parameters** button in the **Parameters** panel. In the **Select model variables** dialog, you can select the parameters you want to estimate in this experiment by checking the box next to it or specifying an expression. For more information see "Select Parameters Using Select Model Variables Dialog Box" on page 2-11.

Select Parameters Using Select Model Variables Dialog Box

Use this dialog box to specify parameters to estimate. The table lists all the variables in the model workspace and the MATLAB workspace that the model uses, whether tunable or not. Select the parameters to estimate by clicking the check box next to each variable. If your model contains a large number of variables, filter the list by typing in the **Filter by variable name** field.

	Variable	Current v...	Used By
☐	freq1	3	engine_idle_speed/Transfer Fcn
☐	freq2	3	engine_idle_speed/Transfer Fcn1
☐	freq3	3	engine_idle_speed/Transfer Fcn2
☐	gain1	100	engine_idle_speed/Transfer Fcn
☐	gain2	0	engine_idle_speed/Transfer Fcn1
☐	gain3	0	engine_idle_speed/Transfer Fcn2

▶ Specify expression indexing if necessary (e.g., a(3) or s.x)

✔ OK   ✖ Cancel   ? Help
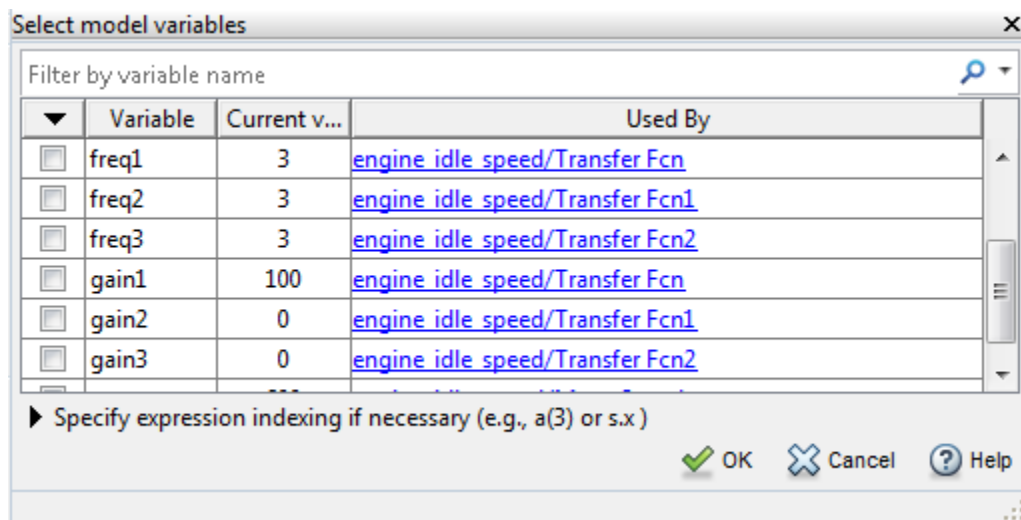
The **Used By** column lists the blocks in the model where each variable is used. When a parameter is used in more than one block, all blocks are listed. To locate blocks in which a parameter is used, click the block name to highlight the block in the model.

If the parameter is not a scalar variable, click **Specify expression indexing if necessary** and enter an expression that resolves to a numeric scalar value.

▼ Specify expression indexing if necessary (e.g., a(3) or s.x)

Non-scalar variables include:

- Simulink software parameter object

  Example: For a Simulink parameter object `k`, enter `k.value`.

- Structure

  Example: For a structure `S`, enter `S.fieldname`, where `fieldname` is the name of the field that contains the parameter.

- Cell array

  Example: Type `C{1}` to select the first element of a cell array `C`.

- MATLAB array

  Example: Type `a(2)` to select the second element of an array `a`.

You cannot use mathematical expressions such as $a + b$. Sometimes, models have parameters that are not explicitly defined in the model itself. For example, a gain `k` could be defined in the MATLAB workspace as `k = a + b`, where `a` and `b` are not defined in the model but `k` is used. To add these independent parameters, such as `k`, to the Select Parameters dialog box, see "Specify Independent Parameters to Estimate" on page 2-14.

Specifying Initial Guesses and Upper/Lower Bounds

After you select parameters, you can specify

- **Initial guess** — The value the estimation uses to start the process.
- **Minimum** — The smallest allowable parameter value. The default is `-Inf`.
- **Maximum** — The largest allowable parameter value. The default is `+Inf`.

  You can enter the initial value in the dialog box below the parameter name. You can specify the minimum and maximum value fields by clicking the arrow . The default minimum and maximum values are `-Inf` and `+Inf`, respectively, but you can select any range you want.

If you believe a parameter lies within a finite range, it is best not to use the default minimum and maximum values. Often, there are computational advantages in specifying finite bounds. It can be very important to specify lower and upper bounds. For example, if a parameter specifies the weight of a part, be sure to specify 0 as the absolute lower bound if better information is unavailable.

Note: When you specify the minimum and maximum values for the parameters, it does not affect your settings in the **Parameters** list under **Data Browser** pane. You make these choices for each experiment.

- **Scale** — Scale is used for normalization, in situations, for example, when parameters are in different units.

Related Examples

- "Specify Independent Parameters to Estimate" on page 2-14
- "Specify Known Initial States" on page 2-18

More About

- "What Is an Experiment?" on page 2-3

Specify Independent Parameters to Estimate

This example shows how to specify independent parameters, that do not appear explicitly in the model, as estimation parameters.

Assume that the parameter Kint in the model srotut1 is related to the parameters x and y according to the relationship Kint=x+y. Also assume that the initial values of x and y are 1 and -0.7 respectively. To estimate x and y instead of Kint, first define these parameters in the model workspace. To do this:

1   At the MATLAB prompt, type

    srotut1

    This opens the srotut1 model window.

2   Select **View** > **Model Explorer** from the srotut1 window to open the Model Explorer window.

3   In the Model Hierarchy tree, select **srotut1** > **Model Workspace**.
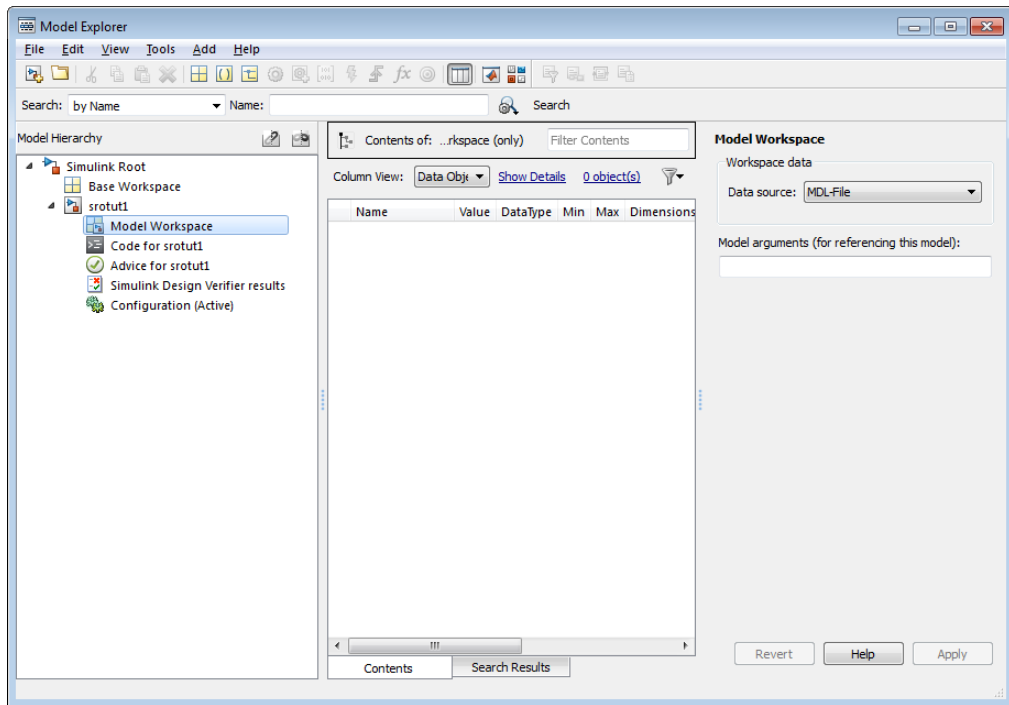
4   Select **Add** > **MATLAB Variable** to add a new variable to the model workspace. A
    new variable with a default name `Var` appears in the **Name** column.

5   Double-click `Var` to make it editable and change the variable name to `x`. Edit the
    initial **Value** to `1`.

6   Repeat steps 4 and 5 to add a variable `y` with an initial value of `-0.7`.

    The Model Explorer window resembles the following figure.
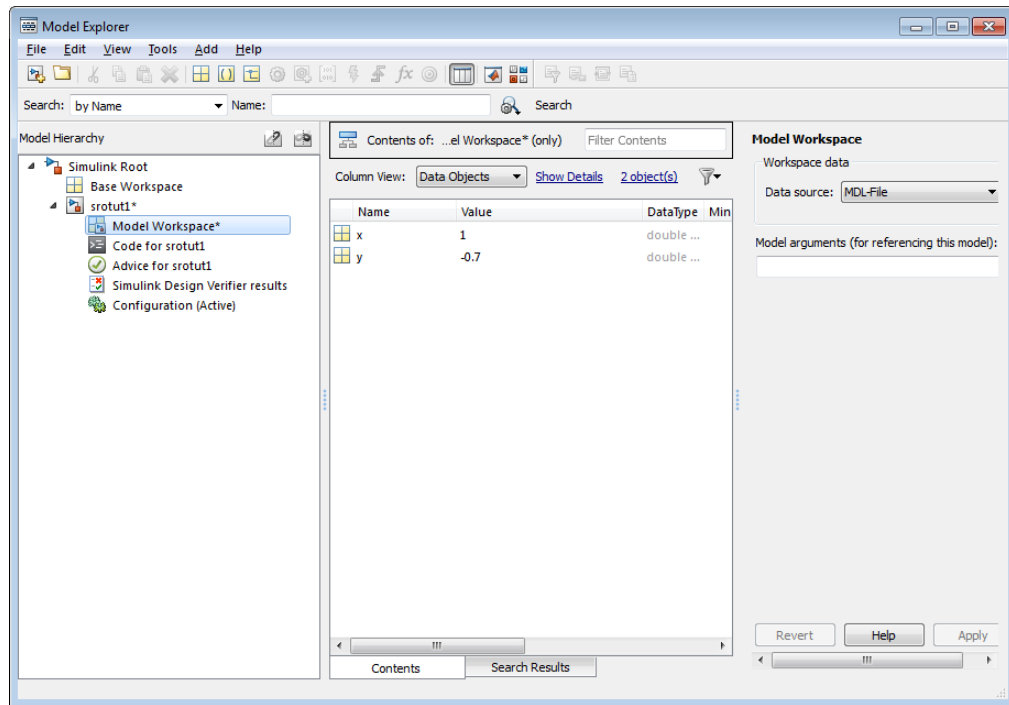
7. To add the Simulation Start function that defines the relationship between `Kint` and the independent parameters `x` and `y`, select **File** > **Model Properties** in the `srotut1` model window.

8. In the Model Properties window, click the **Callbacks** tab.

9. To enter a Simulation start function, select **StartFcn***, and type the name of a new function. For example, `srotut1_start` in the **Simulation start function** panel. Then, click **OK**.

10. Create a MATLAB file named `srotut1_start`. The content of the file defines the relationship between the parameters in the model and the parameters in the workspace. For this example, the content resembles the following:

```
wks = get_param(gcs, 'ModelWorkspace')
x = wks.evalin('x')
y = wks.evalin('y')
Kint = x+y;
```

> **Note:** You must first use the `get_param` function to get the variables x and y from the model workspace before you can use them to define `Kint`.

For x and y to appear in Parameter Estimation tool, the **Select model variables dialog** box, x and y must be used by the model. For how to select parameters for estimation, see "Specify Parameters for Estimation" on page 2-8.

> **Caution** Avoid adding independent parameters together with their corresponding dependent parameters to the lists of parameters to be estimated. Otherwise, the estimation could give incorrect results. For example, when a parameter c depends on the parameters a and b avoid adding all three parameters to the list.

Related Examples

- "Specify Parameters for Estimation" on page 2-8

Specify Known Initial States

In this section...
"When to Specify Initial States Versus Estimate Initial States" on page 2-18
"Specify Model Initial States" on page 2-18

When to Specify Initial States Versus Estimate Initial States

Sets of measured data are often collected at various times and under different initial conditions. When you estimate model parameters using one data set and subsequently run another estimation with a second data set, your parameter values may not match.

You can use the Parameter Estimation tool to estimate the initial conditions using procedures that are similar to those you use to estimate parameters. You can then use these initial condition estimates as a basis for estimating parameters for your Simulink model.

Specify Model Initial States

After you select parameters for estimation, as described in "Specify Parameters for Estimation" on page 2-8, you can specify initial conditions of states in your model. By default, the estimation uses initial conditions specified in the Simulink model. If you want to specify initial conditions other than the defaults, use the **Initial States** panel in the experiment editor dialog. For this example, right click `NewData` and select **Edit...** from the list to open the experiment editor. Then, click **Select Initial States** button.
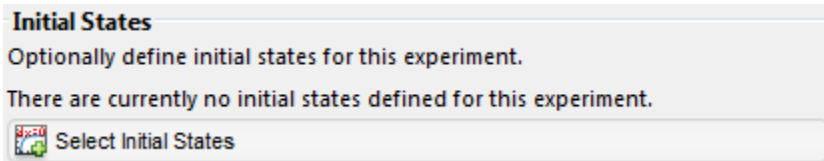
The Select Model States dialog for the `engine_idle_speed` model looks like
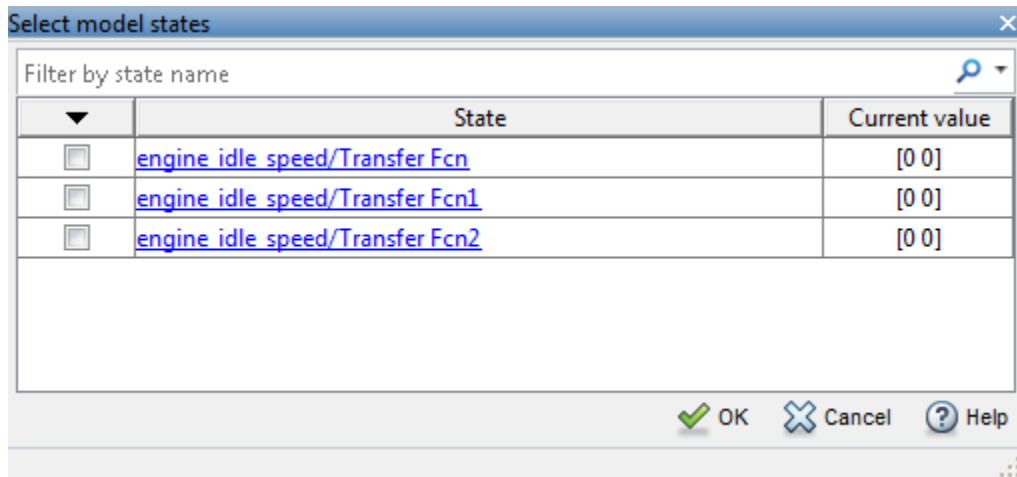
Click the check box next to the state you would like to modify. For example, if you select **engine_idle_speed/Transfer Fcn** and enter the initial values `-0.2` and `0`, the **Initial States** panel now looks like
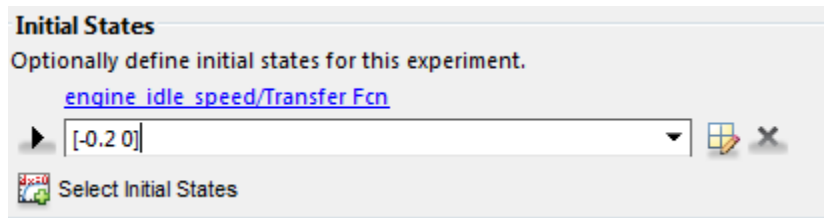
Click **Select Parameters** in the **Parameter Estimation** tab. After you also select the parameters as described in "Specify Parameters for Estimation" on page 2-8, the Edit: Estimated Parameters dialog looks like the following figure.
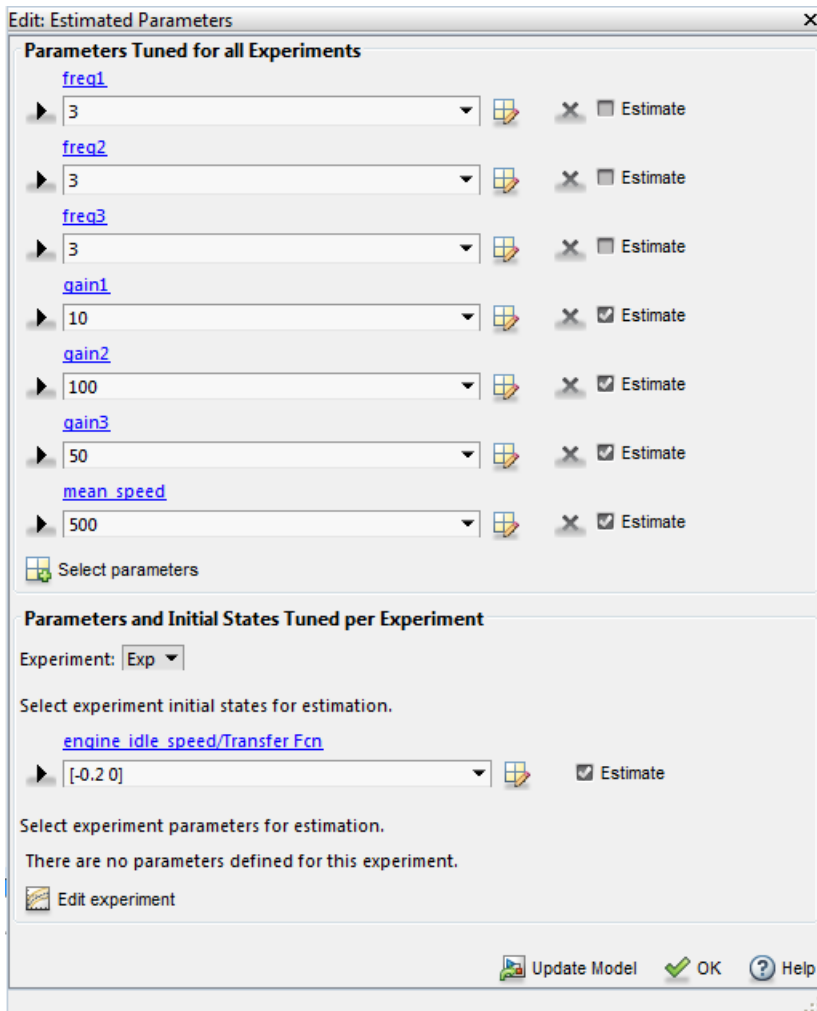
Related Examples

- "Specify Estimation Data" on page 2-4
- "Specify Parameters for Estimation" on page 2-8

More About

- "Experiment Editor" on page 2-22

Experiment Editor

After creating an experiment as in "Create Experiment" on page 2-4, you can launch the experiment editor by right-clicking on the experiment name and selecting **Edit...** from the list. The experiment editor resembles the following figure.
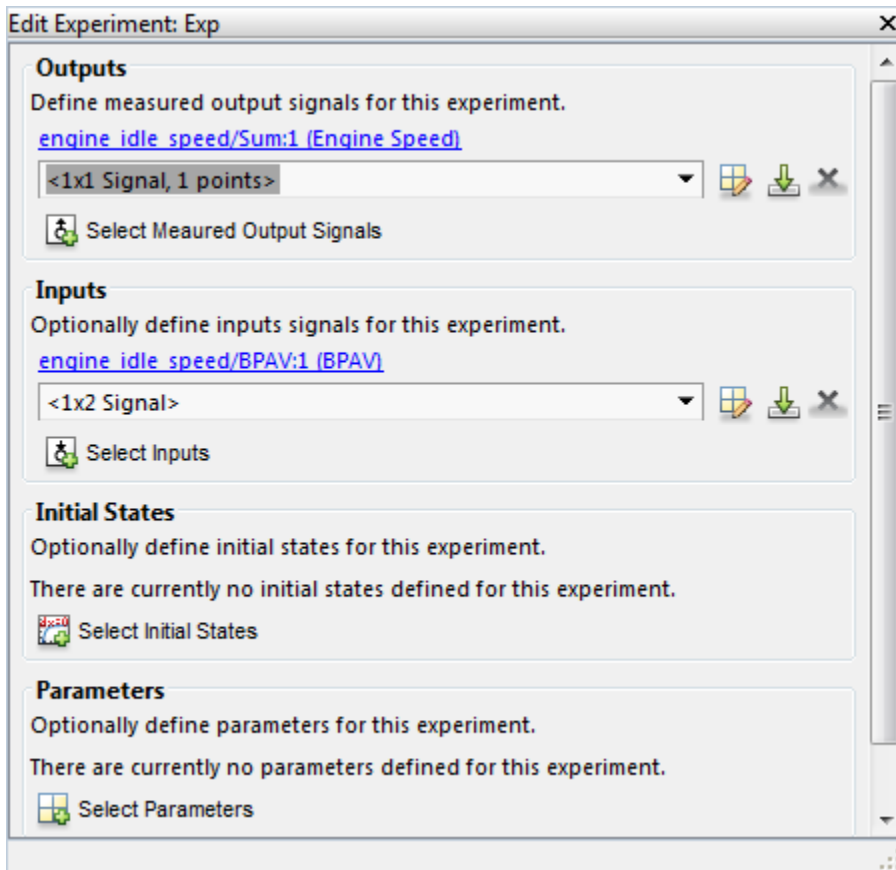
The experiment editor has four panels. You can select output signals and import output data in the **Outputs** panel. You can select input signals and import input data in the **Inputs** panel. You can specify model initial states in the **Initial States** panel. And you can specify parameters to estimate in the **Parameters** panel.

Related Examples

More About

Specify Experiments for Estimation

After specifying the parameters for estimation, select the experiment for the estimation task. In the Parameter Estimation tool, on the **Parameter Estimation** tab, click **Select Experiments**. In the Select Experiments dialog box, you can select the experiment to use for estimation by clicking the check box corresponding to the experiment in the **Estimation** column. For this example, there is only one experiment, `NewData`.
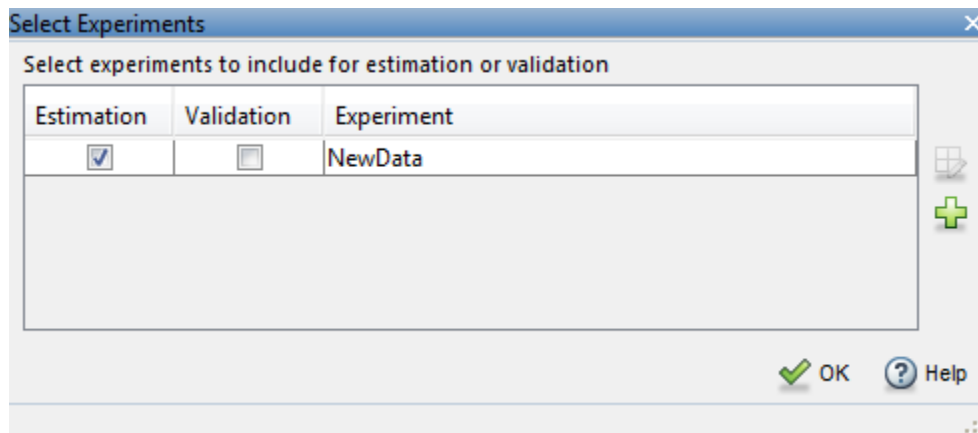
For more information, see "Specify Experiments for Validation" on page 2-48

Related Examples

- "Specify Estimation Data" on page 2-4
- "Specify Parameters for Estimation" on page 2-8
- "Specify Known Initial States" on page 2-18
- "Specify Experiments for Validation" on page 2-48

More About

- "What Is an Experiment?" on page 2-3

Progress Plots

In this section...
"Types of Plots" on page 2-25
"Basic Steps for Creating Plots" on page 2-25

Types of Plots

The following types of plots are available for viewing and evaluating the estimation:

- `Cost function` — Plot the cost function value as it changes during estimation.
- `Measured and simulated` — Plot empirical data against simulated data.
- `Parameter trajectory` — Plot the parameter values as they change.
- `Residuals` — Plot the error between the experimental data and the simulated output.

Basic Steps for Creating Plots

When you run estimation, Parameter Estimation tool automatically displays a parameter trajectory plot that shows the change in the parameter values by iteration. However, you can create other plots for viewing the progress of the estimation before you begin estimating the parameters.

Note: An experiment must be created and selected for estimation before creating views. To learn more, see "Create Experiment" on page 2-4 and "Specify Experiments for Estimation" on page 2-24.

You can plot the measured data by clicking **Add Plot** on the **Parameter Estimation** tab and selecting the experiment to use for estimation under **Experiment Plots** of the drop down list. You can then add the simulated response on top of the measured data plot by clicking **Plot Model Response** on the **Parameter Estimation** tab. Another way of plotting the measured and simulated data is to right-click the experiment name, for this example, `NewData`, and select **Plot measured & simulated data** from the list. The plot looks like this:
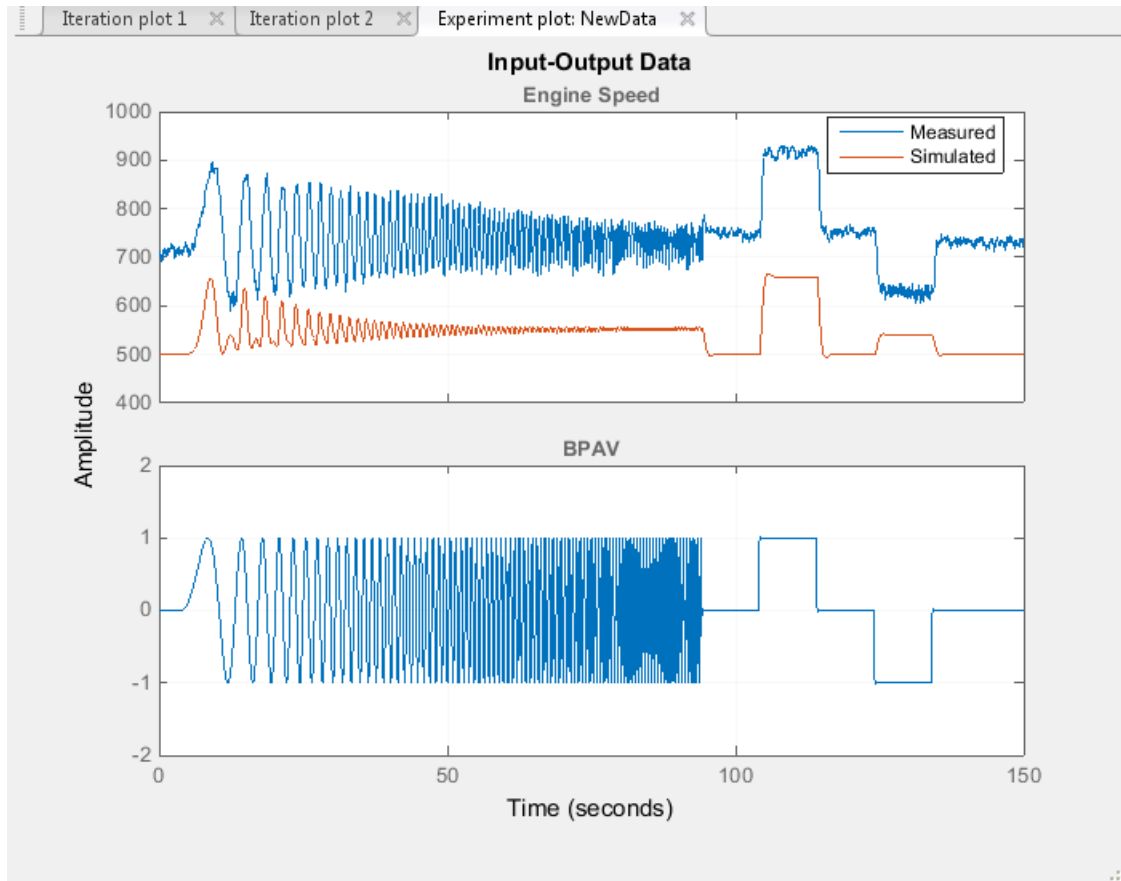
You can see that the measured and simulated data do not match. You can edit the labels, adjust the limits, change the units, and the font style of the plot in the **Property Editor**. To launch the editor, right-click the experiment plot and select **Properties** from the list.
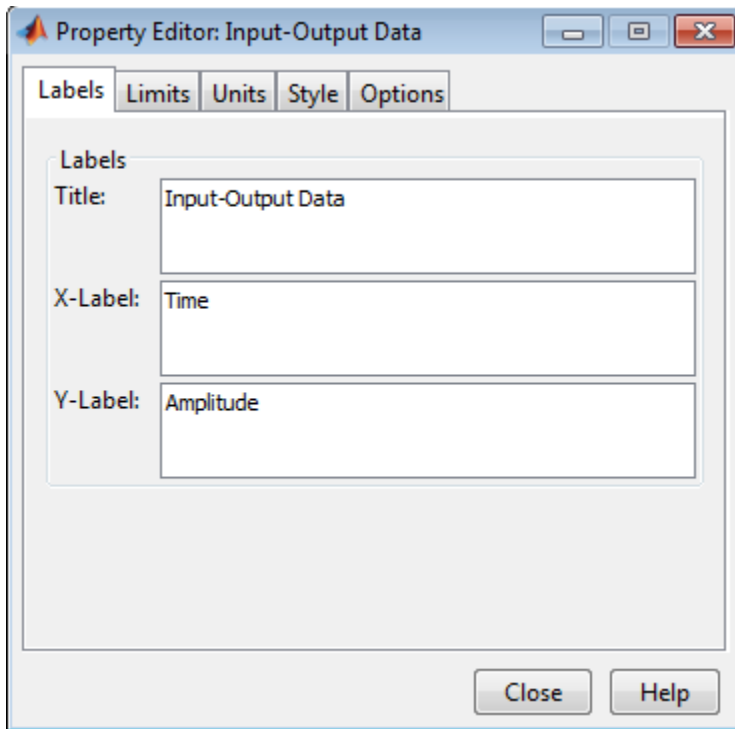
You can also add a plot for the expected estimation cost by clicking **Add Plot** on the Parameter Estimation tab and selecting **Estimation Cost** from the list. You can close the message "`There is no data for ExpCost, run the optimization to update the plot`", by clicking the left arrow toggle to the left of the message. The plot looks like this:
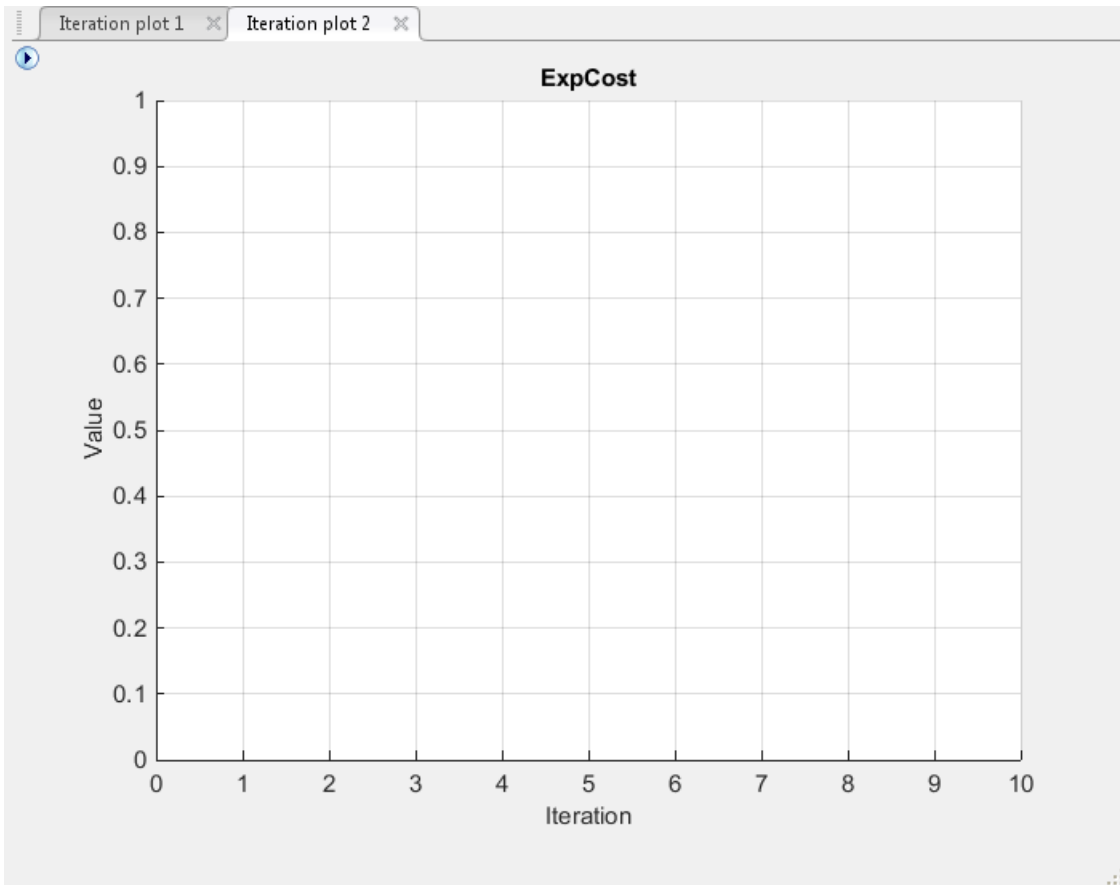
When you right-click the estimated parameters or expected cost plots, you can choose to add the scaled values or save iteration data from the list.
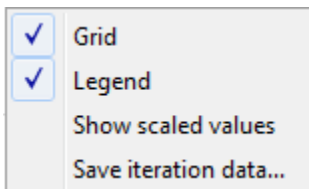
Add the progress plots by clicking the **Add Plot** button on the **Parameter Estimation** tab and selecting the plot in the list. Selecting **Parameter Trajectory** adds the following plot to the tool.
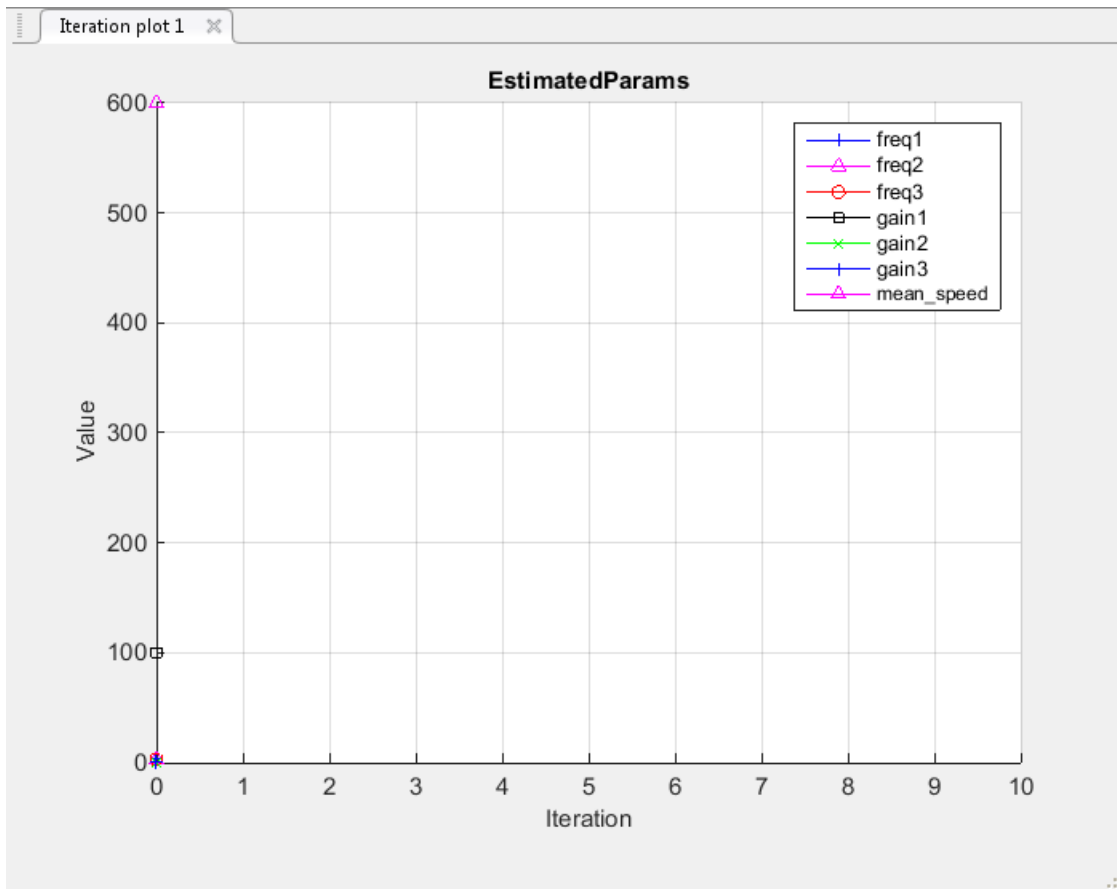
Use the **View** tab of the Parameter Estimation tool to arrange the layout of the plots, so that `Experiment plot: NewData`, `Iteration plot 1`, and `Iteration plot 2` are all visible.
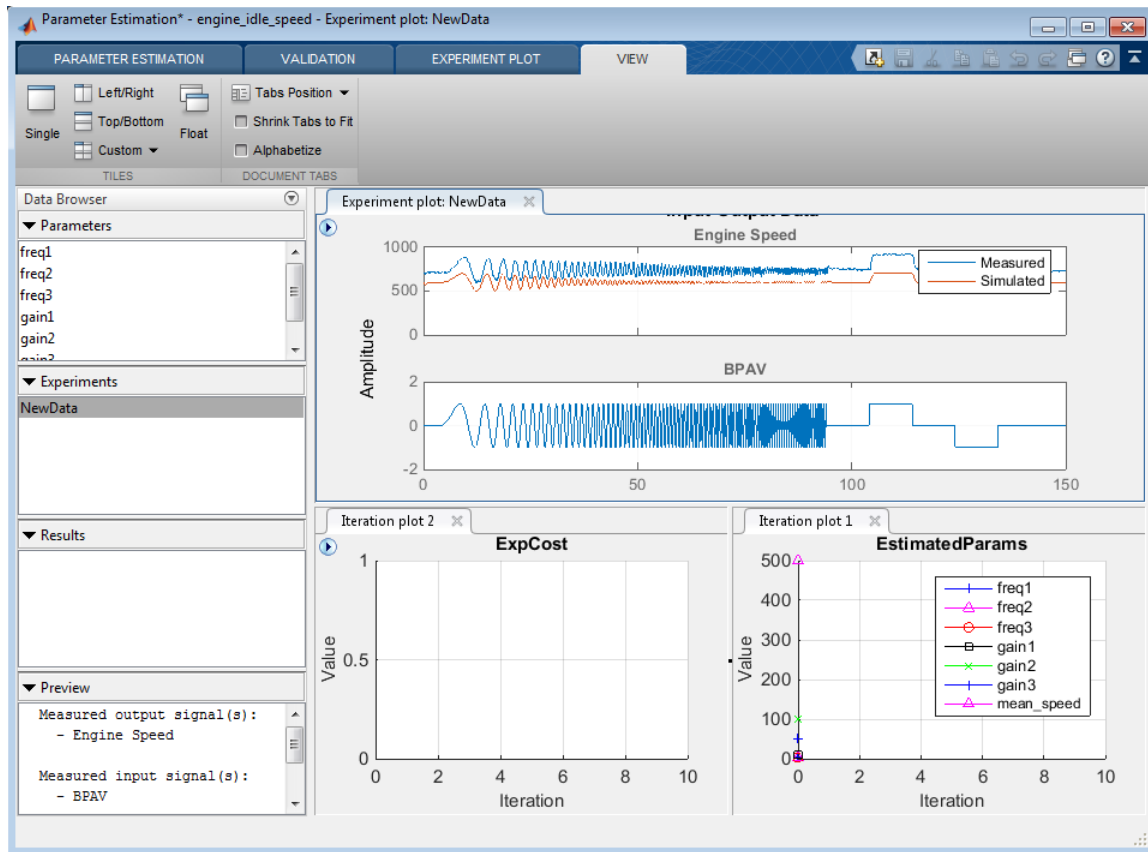
When you perform the estimation, all plots update automatically.

Estimation Options

In this section...
"Access Estimation Options" on page 2-31
"General Options" on page 2-31
"Optimization Options" on page 2-33
"Specify Goodness of Fit Criteria (Cost Function)" on page 2-36
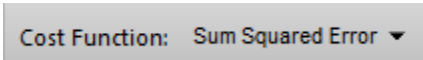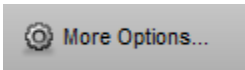
Access Estimation Options

In the Parameter Estimation tool, you can choose the cost function from the **Cost Function** combo box.

Cost Function: Sum Squared Error ▼

You can access other estimation options by clicking the **More Options** button.

⚙ More Options...

Note: Parallel Options (see "Estimate Parameters Using Parallel Computing in the Parameter Estimation Tool" on page 2-64).

General Options

You can set optimization progress and result options for your estimation task in the Estimation Options dialog box.
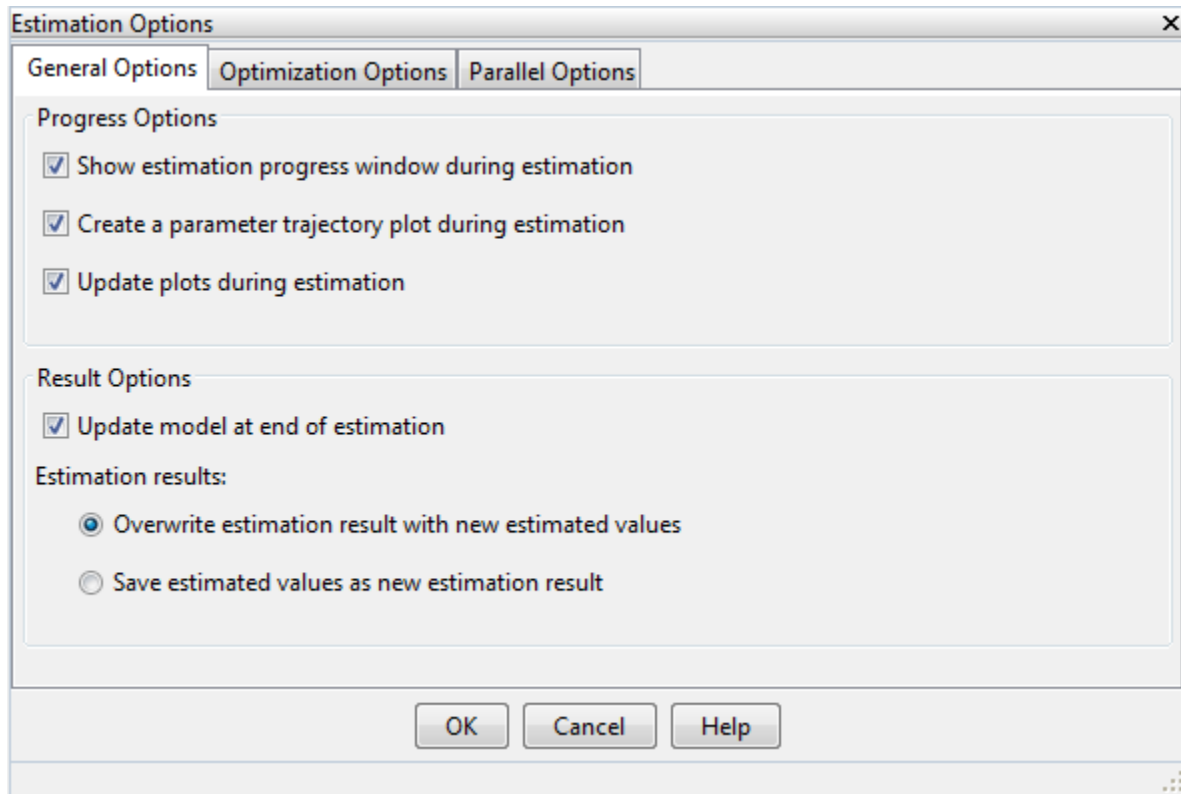
Progress Options

Show estimation progress window during estimation option opens an estimation progress window. The window displays iteration information, such as the cost function, and the termination information, such as whether the optimization converged.

Create a parameter trajectory plot during estimation option creates a plot that shows how the parameter values change by iteration.

Update plots during optimization option updates the progress plots such a cost function and parameter trajectory plot and estimation plots during estimation.

Result Options

Update model at the end of estimation option updates the parameter values in the Simulink model after the estimation terminates.

Overwrite estimation result with new estimated values option overwrites any estimation results or experiments with the new estimated values.

Save estimated values as new estimation result option creates a new variable that contains the estimated parameter values.

Optimization Options

- "Supported Estimation Methods" on page 2-33
- "Optimization Termination Options" on page 2-35
- "Additional Optimization Options" on page 2-35
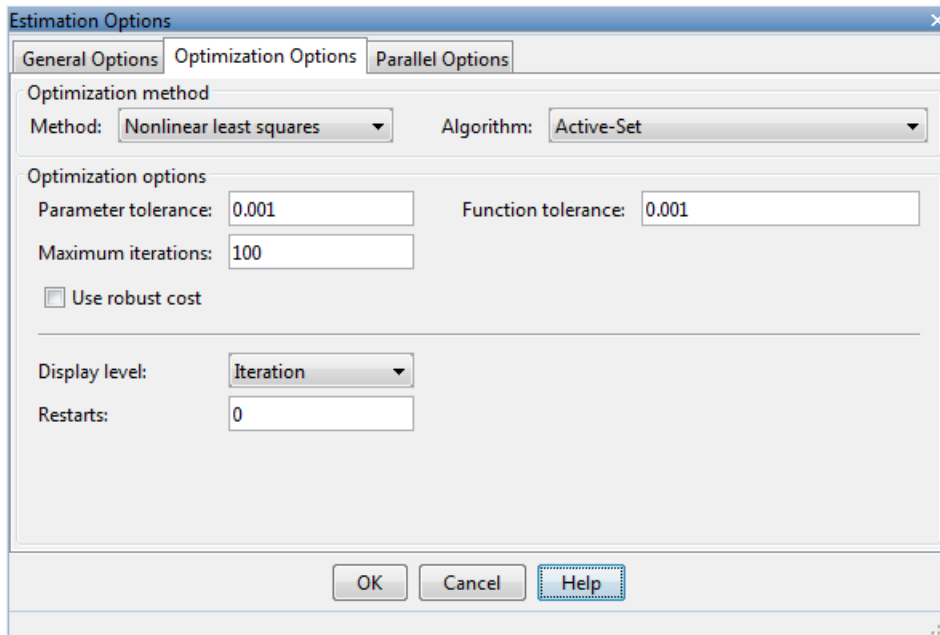
Supported Estimation Methods

Both the **Method** and **Algorithm** options define the optimization method.
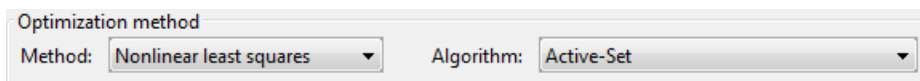
For the **Method** option

- `Nonlinear least squares` (default) — Uses the Optimization Toolbox™ nonlinear least squares function `lsqnonlin`.
- `Gradient descent` — Uses the Optimization Toolbox function `fmincon`.
- `Pattern search` — Uses the pattern search method `patternsearch`. This option requires Global Optimization Toolbox.
- `Simplex search` — Uses the Optimization Toolbox function `fminsearch`, which is a direct search method. `Simplex search` is most useful for simple problems and is sometimes faster than `fmincon` for models that contain discontinuities.

For the `Nonlinear least squares` and `Gradient descent` estimation methods

Method	Algorithm Option	Learn More
`Nonlinear least squares`	 • `Trust-Region-Reflective` (default) • `Levenberg-Marquardt`	In the Optimization Toolbox documentation, see: • "Trust-Region-Reflective Least Squares Algorithm" • "Levenberg-Marquardt Method"
`Gradient descent`	 • `Trust-Region-Reflective` (default) • `Interior-Point` • `Active-Set` • `Sequential Quadratic Programming`	In the Optimization Toolbox documentation, see: • "fmincon Active Set Algorithm" • "fmincon Interior Point Algorithm" • "fmincon Trust Region Reflective Algorithm" • "fmincon SQP Algorithm"

Optimization Termination Options

Several options define when the optimization terminates:

- **Parameter tolerance** — Optimization terminates when successive parameter values change by less than this number.
- **Maximum iterations** — Maximum number of iterations allowed. The optimization terminates when the number of iterations exceeds this value.
- **Function tolerance** — Optimization terminates when successive function values are less than this value.
- **Use robust cost** — Make the optimizer use a robust cost function instead of the default least-squares cost. This is useful if the experimental data has many outliers, or if your data is noisy.

By varying these parameters, you can force the optimization to continue searching for a solution or to continue searching for a more accurate solution.

Additional Optimization Options

- **Display level** — Specify the form of the output that appears in the MATLAB Command Window. The options are:
  - `Iteration` — Display information after each iteration.
  - `None` — Turn off all output.
  - `Notify` — Display output only if the function does not converge.
  - `Final` — Display only the final output.
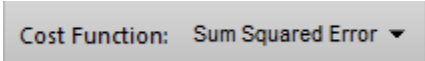
  Refer to the Optimization Toolbox documentation for more information on what type of iterative output each method displays.

- **Restarts** — Indicate the number of times you want to restart, to automatically restart the optimization. In some optimizations, the Hessian may become ill-conditioned and the optimization does not converge. In these cases, it is sometimes useful to restart the optimization after it stops, using the endpoint of the previous optimization as the starting point for the next one.

Specify Goodness of Fit Criteria (Cost Function)

The cost function is a function that estimation methods minimize. You can specify the cost function by selecting from the list in the **Cost Function** combo box on the **Parameter Estimation** tab.

Cost Function: Sum Squared Error ▼

The default cost function is `Sum Squared Error` (sum of squared errors), which uses a least-squares approach. You can also use `Sum-Absolute Error`, the sum of absolute errors.

You can select the robust cost option from the **Optimization Options** tab of the **Estimation Options** editor. To launch the dialog box click **More Options...** on the Parameter Estimation tool. **Use robust cost** option makes the optimizer use a robust cost function instead of the default least-squares cost. This is useful if the experimental data has many outliers, or if your data is noisy.

Progress Display Options

You can specify the display options in the Parameter Estimation tool. On the **Parameter Estimation** tab, in the **Estimate** section, click the arrow , and select `Open Estimation Report`.
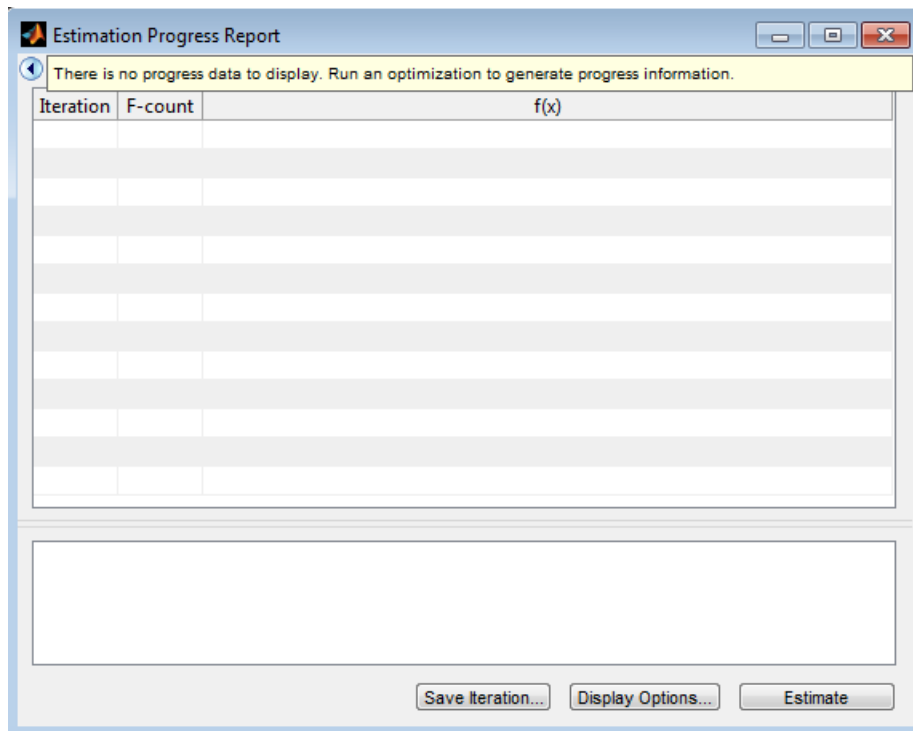
The report displays the iteration number (`Iteration`), the number of times the objective function is calculated (`F-count`), and cost function value (`f(x)`) by default. You can change by clicking **Display Options**.
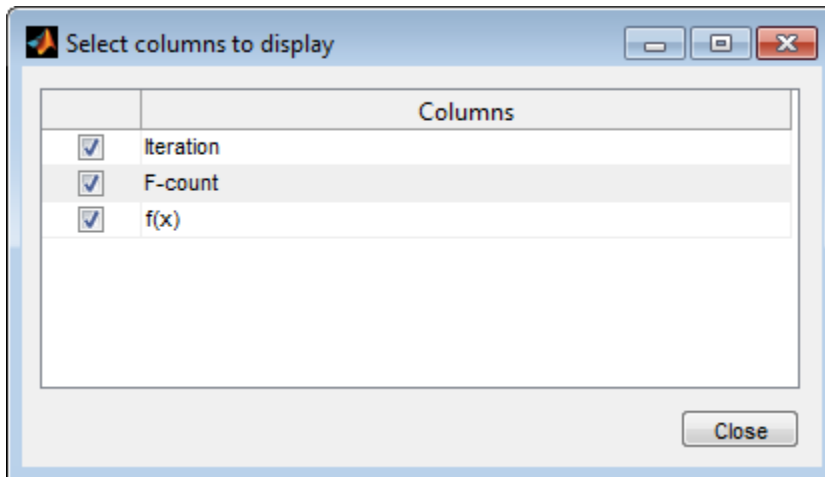
Clear a check box to remove it from the display table. To learn more about the display table, see "Iterative Display" in the Optimization Toolbox documentation.

Run Estimation

Before you begin estimating the parameters, you must have configured the estimation data, selected parameters, and specified estimation options, as described in "Specify Estimation Data" on page 2-4, "Specify Parameters for Estimation" on page 2-8, and "Estimation Options" on page 2-31, respectively.

To start the estimation, in the Parameter Estimation tool, on the **Parameter Estimation** tab, click the **Estimate** button .
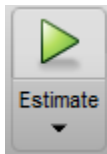
When starting the estimation, a progress window displays. At the end of the estimation, the Estimation Progress Report window should resemble the following:

The estimation results are saved in `EstimatedParams` in the **Results** list on the Browse Data pane.

Right-click `EstimatedParams` and select **Open...** from the menu. The window looks like the following figure.
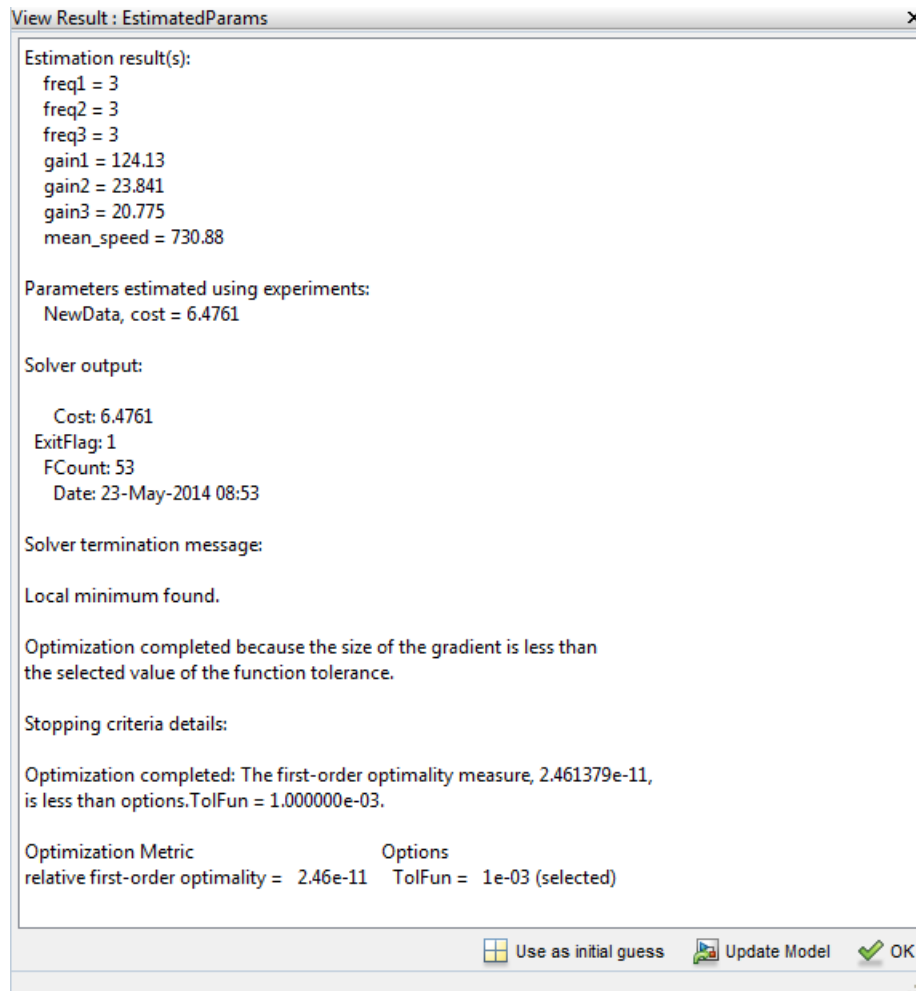
The `EstimatedParams` includes the values of the parameters, the cost function value, and information about the stopping criteria for the estimation. The optimization stops because the successive function values are less than the specified value 1e-3.

The **Estimation Progress Report** includes the change in the cost function in the column titled **NewData(Minimize)**. To see a plot of the change in the cost function

during estimation, add the cost function plot by clicking the **Add Plot** button on the **Parameter Estimation** tab and selecting `Estimation Cost` from the list. After the estimation process completes, the cost function minimization plot appears as shown in the following figure.
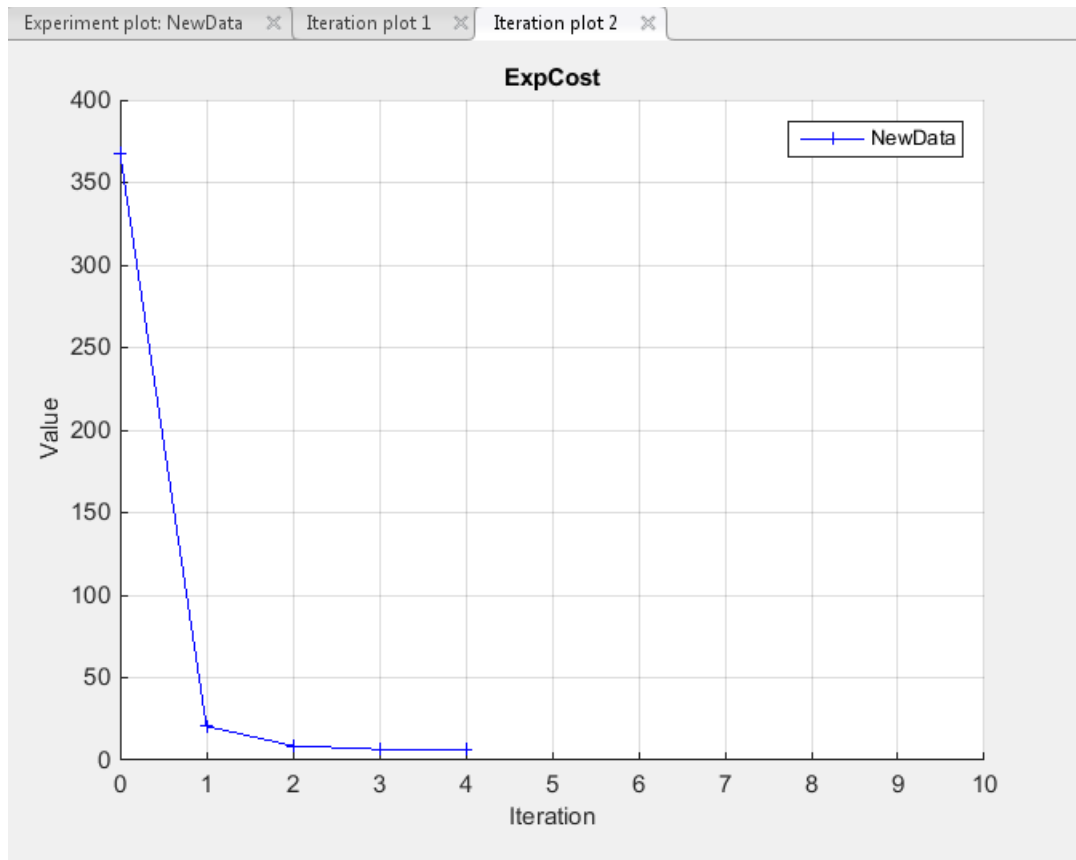
Usually, a lower cost function value indicates a successful estimation, meaning that the experimental data matches the model simulation with the estimated parameters. If the optimization went well, you should see your cost function converge on a minimum value. The lower the cost, the more successful is the estimation.

For information on types of problems you may encounter using optimization solvers, see the following topics in the Optimization Toolbox documentation:

- "When the Solver Fails "
- "When the Solver Might Have Succeeded "
- "When the Solver Succeeds "

The estimated parameters graph shows the change in the estimated value of the parameters by iteration.
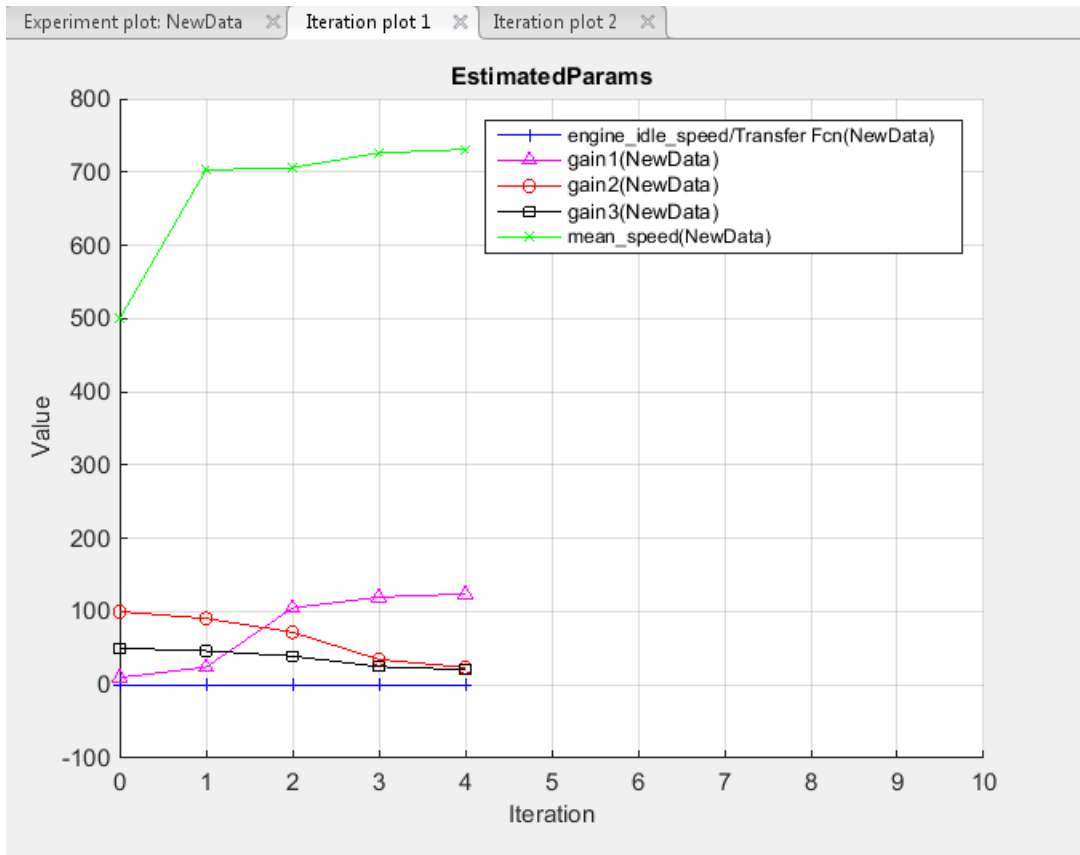
The values of the parameters are recorded with the estimated values.
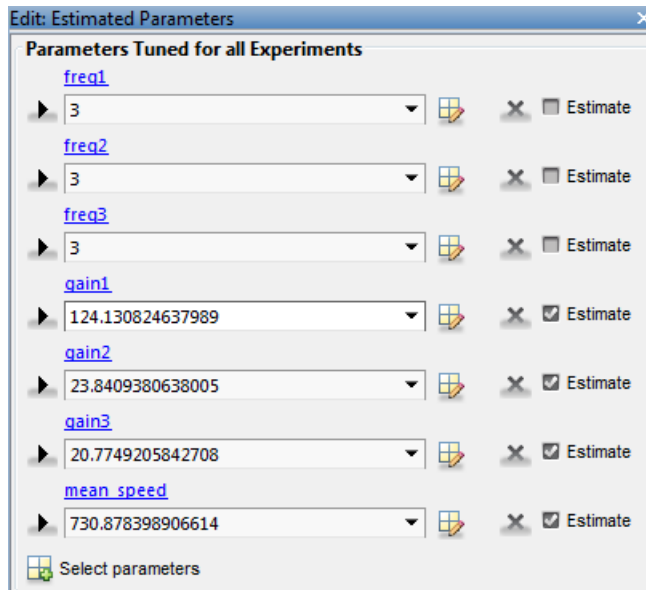
The values of the estimated parameters are also updated in the MATLAB workspace.

You can also examine the measured versus simulated data plot to see how closely the simulated data matches the measured estimation data. The next figure shows the measured versus simulated data plot generated by running the estimation of the engine_idle_speed model (for engine_idle_speed model, see "Create Experiment" on page 2-4). Now, the simulated values match the measured output signal better.
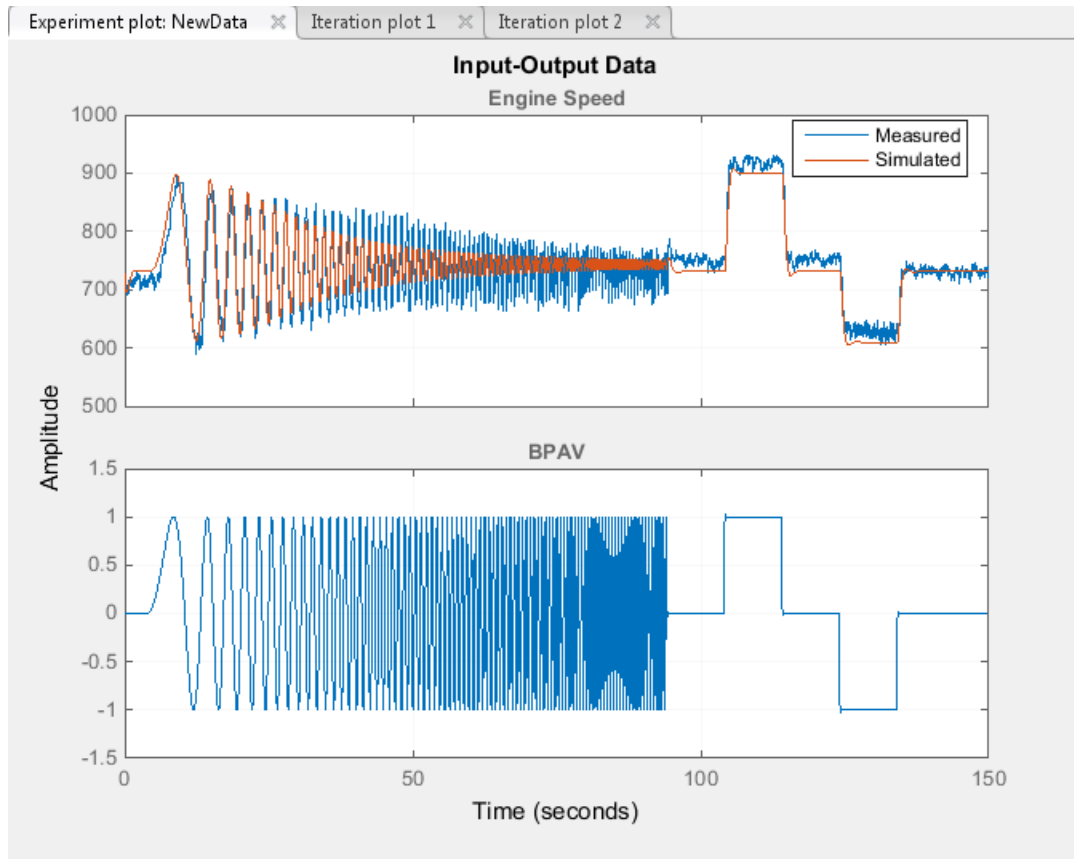
Related Examples

More About

- "What Is an Experiment?" on page 2-3
- "Model Validation" on page 2-46

Model Validation

After you complete estimating parameters as described in "Run Estimation" on page 2-39, validate the results against another set of data.

To validate a model using the Parameter Estimation tool

1   Add a new experiment in the **Experiments** list in the **Data Browser** pane.
2   Import the validation data set to the experiment you want to use for validation. To do this right-click the experiment name and select **Edit...**.
3   Select the experiment for validation.
4   Select the results to use in the validation.
5   Select the plots to display at the end of validation.
6   Run validation and compare the measured validation data against the model output simulated with the estimated parameter to see if they match.

You can run validation after the estimation is complete. Validations can use other validation data sets for comparison with the model response. You must set up all estimation plots before an estimation, and you can watch the views update in real time. Validations appear after you have completed an estimation and do not update.

You can validate data by comparing measured and simulated data for your estimation and validation experiments.

Also, it is useful to compare residuals of measured and simulated data in the same way.

Related Examples

- "Load and Import Validation Data" on page 2-47
- "Specify Experiments for Validation" on page 2-48
- "Select Results for Validation" on page 2-50
- "Compare Measured and Simulated Responses" on page 2-54

More About

- "What Is an Experiment?" on page 2-3

Load and Import Validation Data

To validate the estimated parameters computed in "Run Estimation" on page 2-39, import the data into the experiment you want to use for validation in the Parameter Estimation tool.

At the MATLAB prompt, load the validation data into the MATLAB workspace by typing

```
load iodataval
```

Import this data into the validation experiment in the Parameter Estimation tool.

1   Add a new experiment in the **Experiments** list in the **Data Browser** pane. You can rename the experiment by right-clicking and selecting **Rename** from the list.

2   Right click the experiment name and select **Edit...**

3   Type `[time,iodataval(:,1)]` in the dialog box in the **Inputs** panel.

4   Type `[time,iodataval(:,2)]` in the dialog box in the **Outputs** panel.

Related Examples

-   "Import Data" on page 1-6
-   "Specify Experiments for Validation" on page 2-48
-   "Select Results for Validation" on page 2-50
-   "Compare Measured and Simulated Responses" on page 2-54

More About

-   "Model Validation" on page 2-46

Specify Experiments for Validation

After importing data into an experiment that you want to use for validation, select the experiment for the validation task. In the Parameter Estimation tool, on the **Validation** tab, click **Select Experiments**.
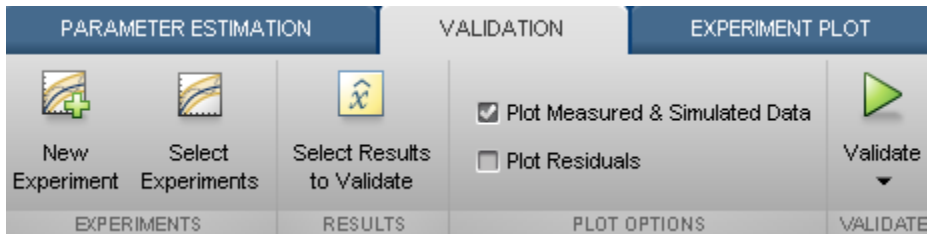
When you create an experiment, it is by default selected for estimation. To select an experiment for validation in the Select Experiments dialog box, deselect the check box under the **Estimation** column and select the check box under the **Validation** column for the corresponding experiment.
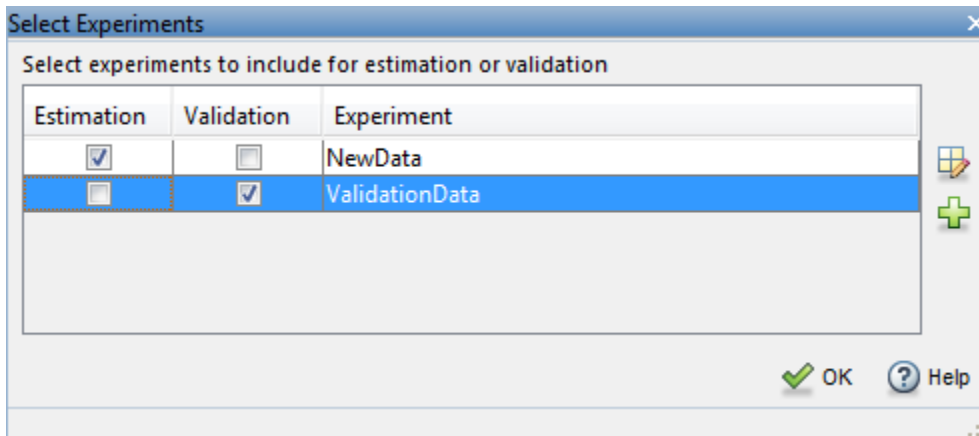
Related Examples

- "Load and Import Validation Data" on page 2-47
- "Select Results for Validation" on page 2-50
- "Compare Measured and Simulated Responses" on page 2-54

More About

- "Model Validation" on page 2-46

Select Results for Validation

After you run the estimation as described in "Run Estimation" on page 2-39, the results are recorded in `EstimatedParams` by default. After you import the validation data, as described in "Load and Import Validation Data" on page 2-47, you must make sure the validation uses the tuned parameter values in `EstimatedParams` in validation. Validation Tab lets you do this. In the Parameter Estimation tool, on the **Validation** tab, click **Select Results to Validate**.
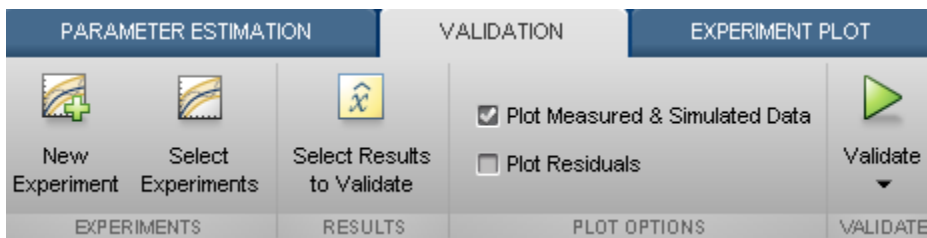
In Select Results dialog box, validate the estimated parameters by selecting `EstimatedParams` and deselecting `Use current parameter values`.
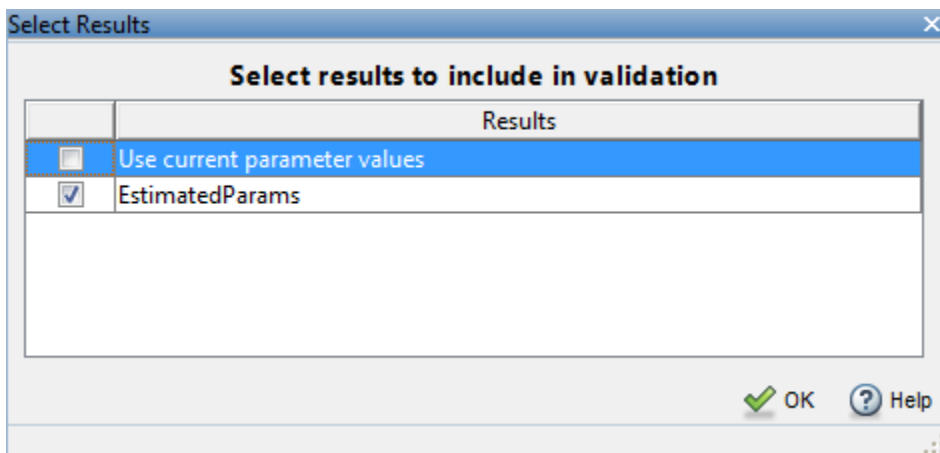
Related Examples

- "Load and Import Validation Data" on page 2-47
- "Specify Experiments for Validation" on page 2-48

- "Compare Measured and Simulated Responses" on page 2-54

More About

- "Model Validation" on page 2-46

Select Plots and Run Validation

Parameter Estimation tool displays the measured and simulated responses and the residuals plot at the end of validation. Select the plots to display by checking the corresponding box on the **Validation Tab**.
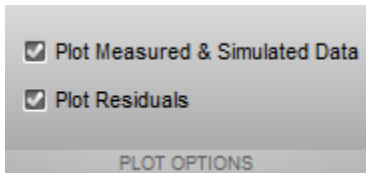
To perform the validation, on the **Validation** tab, click **Validate**. The Validation Progress Report shows the status of the validation.
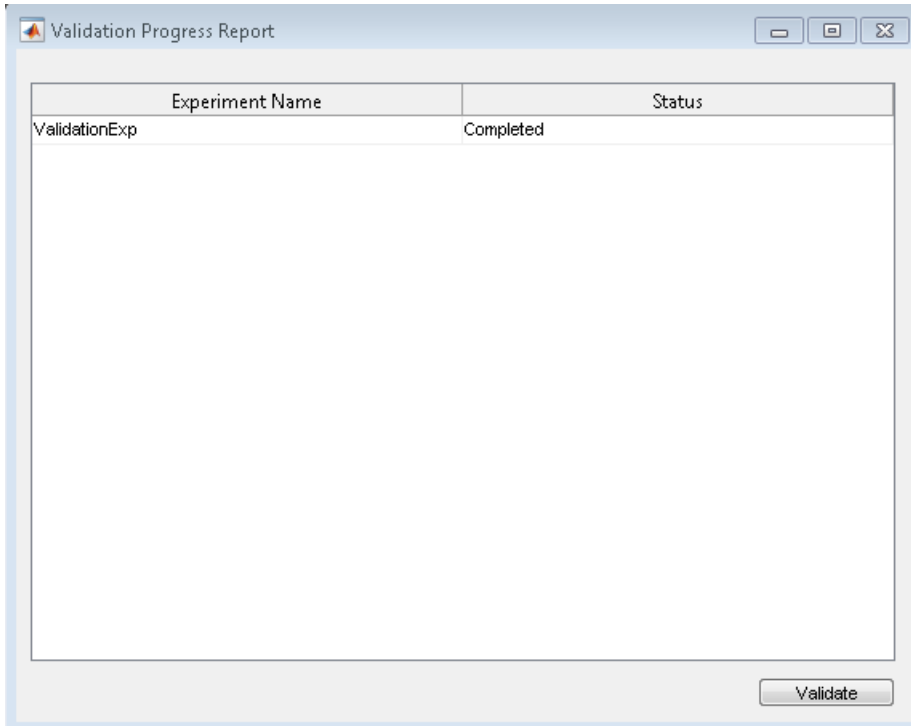
Related Examples

More About

Compare Measured and Simulated Responses

Compare the measured and simulated responses using the experiment plot and residuals plot. To create these plots, select the corresponding check boxes on the **Validation Tab** of the Parameter Estimation tool before you start validation.
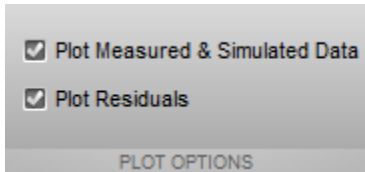
Experiment Plot

The Parameter Estimation tool by default displays the experiment plot for each experiment selected for validation. Each experiment plot shows the measured data, as well as data from simulation using each set of results selected. For example, the following figure shows the experiment plot for the `ValidationExp` data.
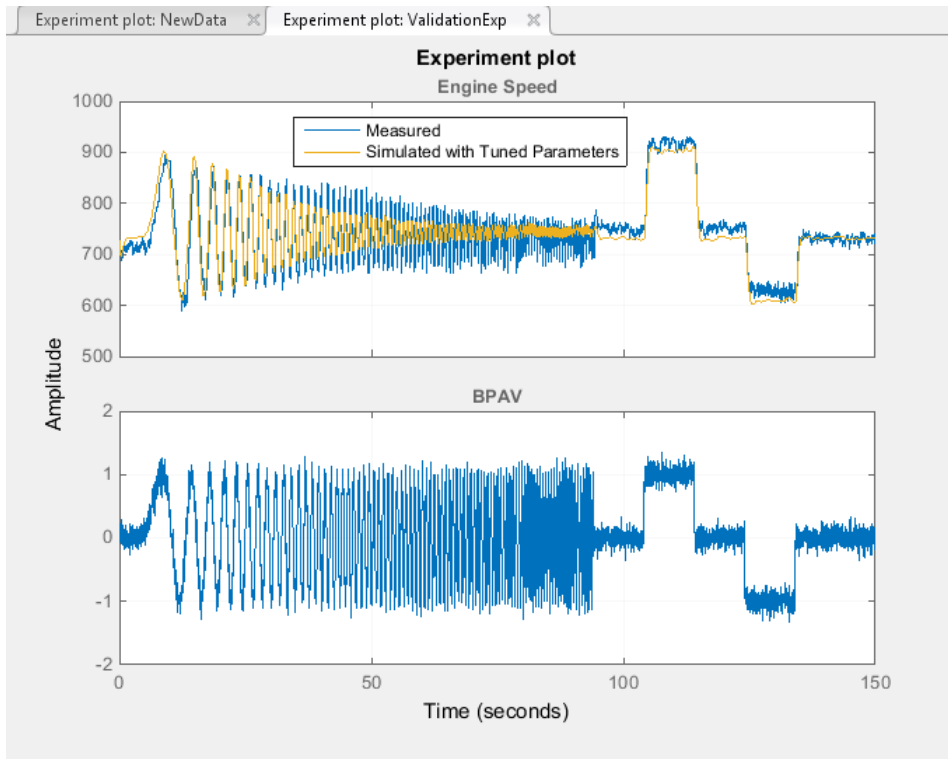
Residuals Plot

After comparing the measured and simulated responses for an estimation, examine the residuals. The residuals plot shows the difference between the simulated response and measured data. To indicate a good fit between the simulated output and measured data, the residuals should:

- Lie within a small percent of the maximum output variation.

- Not display any systematic patterns.

For example, you can see from the following figure that the residuals for `ValidationExp` data satisfy both criteria.
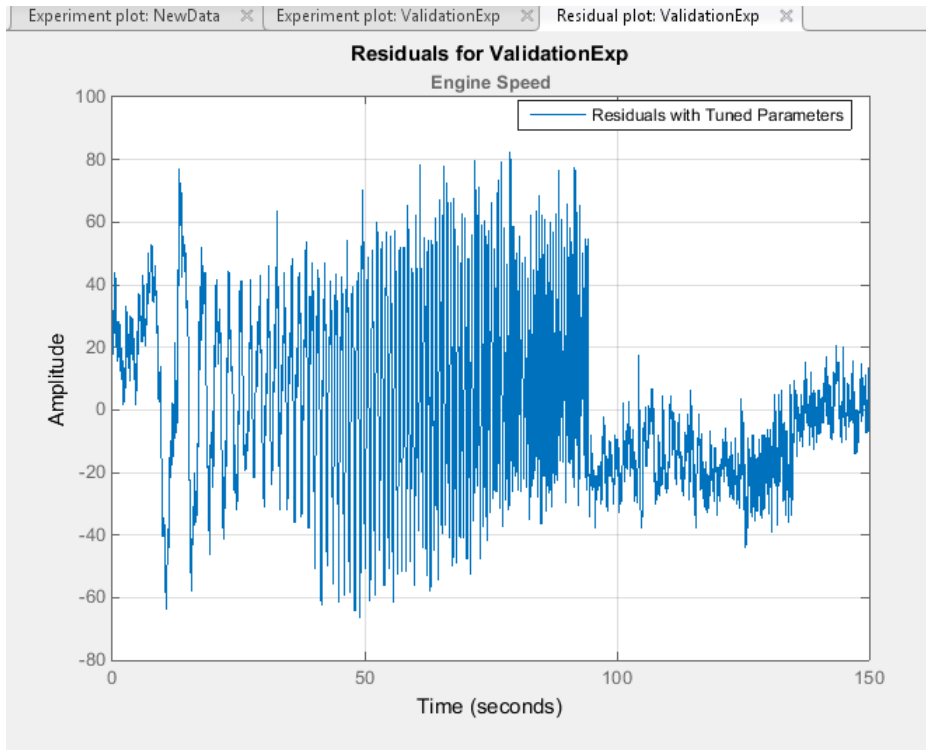
Related Examples

- "Load and Import Validation Data" on page 2-47
- "Specify Experiments for Validation" on page 2-48
- "Select Results for Validation" on page 2-50

More About

- "Model Validation" on page 2-46

Accelerating Model Simulations During Estimation

In this section...
"About Accelerating Model Simulations During Estimation" on page 2-57
"Limitations" on page 2-57
"Setting the Accelerator Mode for Parameter Estimation" on page 2-57

About Accelerating Model Simulations During Estimation

You can accelerate the parameter estimation computations by changing the simulation mode of your Simulink model. Simulink Design Optimization software supports `Normal` and `Accelerator` simulation modes. For more information about these modes, see "How Acceleration Modes Work" in the Simulink documentation.

The default simulation mode is `Normal`. In this mode, Simulink software uses interpreted code, rather than compiled C code during simulations.

In the `Accelerator` mode, Simulink Design Optimization software runs simulations during estimation with compiled C code. Using compiled C code speeds up the simulations and reduces the time to estimate parameters.

Limitations

You cannot use the `Accelerator` mode if your model contains algebraic loops. If the model contains MATLAB function blocks, you must either remove them or replace them with Fcn blocks.

Setting the Accelerator Mode for Parameter Estimation

To set the simulation mode to `Accelerator`, open the Simulink model window and perform one of the following actions:

- Select **Simulation** > **Mode** > **Accelerator**.
- Choose `Accelerator` from the drop-down list as shown in the next figure.
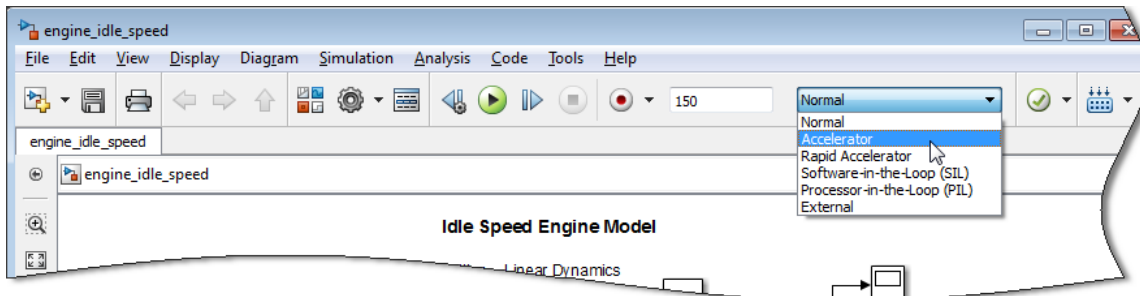
Tip To obtain the maximum performance from the `Accelerator` mode, close all Scope blocks in your model.

Speedup Parameter Estimation Using Parallel Computing

When to Use Parallel Computing for Parameter Estimation

You can use Simulink Design Optimization software with Parallel Computing Toolbox™ software to speed up parameter estimation of Simulink models. Using parallel computing may reduce the estimation time in the following cases:

- The model contains a large number parameters to estimate, and the estimation method is specified as either `Nonlinear least squares` or `Gradient descent`.
- The `Pattern search` method is selected as the estimation method.
- The model is complex and takes a long time to simulate.

When you use parallel computing, the software distributes independent simulations to run them in parallel on multiple MATLAB sessions, also known as *workers*. The time required to simulate the model dominates the total estimation time. Therefore, distributing the simulations significantly reduces the estimation time.

For information on how the software distributes the simulations and the expected speedup, see "How Parallel Computing Speeds Up Estimation" on page 2-59.

For information on configuring your system and using parallel computing, see "How to Use Parallel Computing for Parameter Estimation" on page 2-63.

How Parallel Computing Speeds Up Estimation

You can enable parallel computing with the `Nonlinear least squares`, `Gradient descent` and `Pattern search` estimation methods. The following sections describes the potential speedup using parallel computing for estimation:

- "Parallel Computing with Nonlinear least squares and Gradient descent Methods" on page 2-60
- "Parallel Computing with the Pattern search Method" on page 2-61

Parallel Computing with Nonlinear least squares and Gradient descent Methods

When you select `Gradient descent` as the estimation method, the model is simulated during the following computations:

- Objective value computation — One simulation per iteration
- Objective gradient computations — Two simulations for every tuned parameter per iteration
- Line search computations — Multiple simulations per iteration

The total time, T_{total}, taken per iteration to perform these simulations is given by the following equation:

$$T_{total} = T + (N_p \times 2 \times T) + (N_{ls} \times T) = T \times (1 + (2 \times N_p) + N_{ls})$$

where T is the time taken to simulate the model and is assumed to be equal for all simulations, N_p is the number of parameters to estimate, and N_{ls} is the number of line searches. N_{ls} is difficult to estimate and you generally assume it to be equal to one, two, or three.

When you use parallel computing, the software distributes the simulations required for objective gradient computations. The simulation time taken per iteration when the gradient computations are performed in parallel, T_{totalP}, is approximately given by the following equation:

$$T_{totalP} = T + \left(ceil\left(\frac{N_p}{N_w}\right) \times 2 \times T\right) + (N_{ls} \times T) = T \times \left(1 + 2 \times ceil\left(\frac{N_p}{N_w}\right) + N_{ls}\right)$$

where N_w is the number of MATLAB workers.

Note: The equation does not include the time overheads associated with configuring the system for parallel computing and loading Simulink software on the remote MATLAB workers.

The expected reduction of the total estimation time is given by the following equation:

$$\frac{T_{totalP}}{T_{total}} = \frac{1 + 2 \times ceil\left(\dfrac{N_p}{N_w}\right) + N_{ls}}{1 + (2 \times N_p) + N_{ls}}$$

For example, for a model with N_p=3, N_w=4, and N_{1s}=3, the expected reduction of the total

estimation time equals $\dfrac{1 + 2 \times ceil\left(\dfrac{3}{4}\right) + 3}{1 + (2 \times 3) + 3} = 0.6$.

Parallel Computing with the Pattern search Method

The `Pattern search` method uses search and poll sets to create and compute a set of candidate solutions at each estimation iteration.

The total time, T_{total} , taken per iteration to perform these simulations, is given by the following equation:

$$T_{total} = (T \times N_p \times N_{ss}) + (T \times N_p \times N_{ps}) = T \times N_p \times (N_{ss} + N_{ps})$$

where T is the time taken to simulate the model and is assumed to be equal for all simulations, N_p is the number of parameters to estimate, N_{ss} is a factor for the search set size, and N_{ps} is a factor for the poll set size. N_{ss} and N_{ps} are typically proportional to N_p .

When you use parallel computing, Simulink Design Optimization software distributes the simulations required for the search and poll set computations, which are evaluated in separate parfor loops. The simulation time taken per iteration when the search and poll sets are computed in parallel, T_{totalP} , is given by the following equation:

$$T_{totalP} = (T \times ceil(N_p \times \frac{N_{ss}}{N_w})) + (T \times ceil(N_p \times \frac{N_{ps}}{N_w}))$$
$$= T \times (ceil(N_p \times \frac{N_{ss}}{N_w}) + ceil(N_p \times \frac{N_{ps}}{N_w}))$$

where N_w is the number of MATLAB workers.

Note: The equation does not include the time overheads associated with configuring the system for parallel computing and loading Simulink software on the remote MATLAB workers.

The expected speed up for the total estimation time is given by the following equation:

$$\frac{T_{totalP}}{T_{total}} = \frac{ceil(N_p \times \frac{N_{ss}}{N_w}) + ceil(N_p \times \frac{N_{ps}}{N_w})}{N_p \times (N_{ss} + N_{ps})}$$

For example, for a model with N_p=3, N_w=4, N_{ss}=15, and N_{ps}=2, the expected speedup

equals $\dfrac{ceil(3 \times \frac{15}{4}) + ceil(3 \times \frac{2}{4})}{3 \times (15 + 2)} = 0.27$.

Using the `Pattern search` method with parallel computing may not speed up the estimation time. When you do not use parallel computing, the method stops searching for a candidate solution at each iteration as soon as it finds a solution better than the current solution. When you use parallel computing, the candidate solution search is more comprehensive. Although the number of iterations may be larger, the estimation without using parallel computing may be faster.

Related Examples

How to Use Parallel Computing for Parameter Estimation

In this section...

Configure Your System for Parallel Computing

You can speed up parameter estimation using parallel computing on multicore processors or multiprocessor networks. Use parallel computing with the Parameter Estimation tool and `sdo.optimize` to estimate parameters using the `fmincon`, `lsqonlin`, and `patternsearch` methods. Parallel computing is not supported for the `fminsearch` (**Simplex search**) method.

When you estimate model parameters using parallel computing, the software uses the available parallel pool. If none is available, and you select **Automatically create a parallel pool** in your Parallel Computing Toolbox preferences, the software starts a parallel pool using the settings in those preferences. To open a parallel pool that uses a specific cluster profile, use:

```
parpool(MyProfile);
```

`MyProfile` is the name of a cluster profile.

For information regarding creating a cluster profile, see "Create and Modify Cluster Profiles" in the Parallel Computing Toolbox documentation.

Model Dependencies

Model dependencies are any referenced models, data such as model variables, S-functions, and additional files necessary to run the model. Before starting the optimization, verify that the model dependencies are complete. Otherwise, you may get unexpected results.

Making Model Dependencies Accessible to Remote Workers

When you use parallel computing, the Simulink Design Optimization software helps you identify model dependencies. To do so, the software uses the Simulink Manifest Tools. The dependency analysis may not find all the files required by your model. To learn more, see "Scope of Dependency Analysis" in the Simulink documentation. If your model has dependencies that are undetected or inaccessible by the parallel pool workers, then add them to the list of model dependencies.

The dependencies 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.
- Path dependencies: the paths to the model dependencies are specified to the parallel pool workers. If you are working in a multi-platform scenario, ensure that the paths are compatible across platforms.

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.

For more information, see:

- "Estimate Parameters Using Parallel Computing in the Parameter Estimation Tool" on page 2-64
- "Estimate Parameters Using Parallel Computing (Code)" on page 2-67

Estimate Parameters Using Parallel Computing in the Parameter Estimation Tool

To estimate model parameters using parallel computing in the Parameter Estimation tool:

1 Ensure that the software can access parallel pool workers that use the appropriate cluster profile.

   For more information, see "Configure Your System for Parallel Computing" on page 2-63.

2 Open the Parameter Estimation tool for the Simulink model.

3 Configure the estimation data, estimation parameters and states, and, optionally, estimation settings.

For more information, see "Specify Estimation Data" on page 2-4, "Specify Parameters for Estimation" on page 2-8, and "Optimization Options" on page 2-33.

4

On the **Parameter Estimation** tab, click ⚙**More Options** to open the **Estimation Options** dialog box.

5 Select the **Parallel Options** tab.

6 Select the **Use the parallel pool during optimization** check box.

This option checks for dependencies in your Simulink model. The file dependencies are displayed in the **Model file dependencies** list box, and corresponding path to the files in **Model path dependencies**. The files listed in **Model file dependencies** are copied to the remote workers.

Note: The automatic dependencies check may not detect all the dependencies in your model.

For more information, see "Model Dependencies" on page 2-63. In this case, add the undetected dependencies manually.

7 Add any file dependencies that the automatic check does not detect.

Specify the files in the **Model file dependencies** list box separated by semicolons or on separate lines.

Alternatively, click **Add file dependency** to open a dialog box, and select the file to add.

Note: If you do not want to copy the files to the remote workers, delete all entries in the **Model file dependencies** list box. Populate the **Model path dependencies** list box by clicking the **Sync path dependencies from model**, and add any undetected path dependencies. In addition, in the list box, update the paths on local drives to make them accessible to remote workers. For example, change `C:\` to `\\\hostname\\C$\\`.

---

8 If you modify the Simulink model, resync the dependencies to ensure that any new dependencies are detected. Click **Sync file dependencies from model** in the **Parallel Options** tab to rerun the automatic dependency check for your model.

This action updates the **Model file dependencies** list box with any new file dependency found in the model.

9 Click **OK**.

10 In the **Parameter Estimation** tab, click **Estimate** to estimate the model parameters using parallel computing.

For information on troubleshooting problems related to estimation using parallel computing, see "Troubleshooting" on page 2-68.

Estimate Parameters Using Parallel Computing (Code)

To use parallel computing for parameter estimation at the command line:

1 Ensure that the software can access parallel pool workers that use the appropriate cluster profile.

For more information, see "Configure Your System for Parallel Computing" on page 2-63.

2 Open the model.

3 Configure an estimation experiment. For example, see "Estimate Model Parameter Values (Code)".

4 Enable parallel computing using an optimization option set, `opt`.

```
opt = sdo.OptimizeOptions;
opt.UseParallel = 'always';
```

5 Find the model dependencies.

```
[dirs, files] = sdo.getModelDependencies(modelname)
```

Note: `sdo.getModelDependencies` may not detect all the dependencies in your model. For more information, see "Model Dependencies" on page 2-63. In this case, add the undetected dependencies manually.

6 Modify `files` to include any file dependencies that `sdo.getModelDependencies` does not detect.

```
files = vertcat(files,'C:\matlab\work\filename.m')
```

Note: If you do not want to copy the files to the remote workers, use the path dependencies. Add any undetected path dependencies to `dirs` and update the paths on local drives to make them accessible to remote workers. See `sdo.getModelDependencies` for more details.

---

7 Add the file dependencies for optimization.

```
opt.ParallelFileDependencies = files;
```

8 Run the optimization.

```
[pOpt,opt_info] = sdo.optimize(opt_fcn,param,opt);
```

For information on troubleshooting problems related to estimation using parallel computing, see "Troubleshooting" on page 2-68.

Troubleshooting

- "Why Are the Estimation Results With and Without Parallel Computing Different?" on page 2-68
- "Why Didn't the Estimation Speed up Using Parallel Computing?" on page 2-68
- "Why Doesn't the Estimation Using Parallel Computing Make Any Progress?" on page 2-69
- "Why Does the Estimation Using Parallel Computing Continue When I Click Stop?" on page 2-69

Why Are the Estimation Results With and Without Parallel Computing Different?

- Different numerical precision on the client and worker machines can produce marginally different simulation results. Thus, the optimization method can take a different solution path and produce a different result.
- When you use parallel computing with the `Pattern search` method, the search is more comprehensive and can result in a different solution. To learn more, see "Parallel Computing with the Pattern search Method" on page 2-61.

Why Didn't the Estimation Speed up Using Parallel Computing?

- When you estimate a few parameters or when the model does not take long to simulate, you do not see a speedup in the estimation time. In such cases, the overhead

associated with creating and distributing the parallel tasks outweighs the benefits of running the estimation in parallel.

- Using the `Pattern search` method with parallel computing might not speed up the optimization time. Without parallel computing, the method stops the search at each iteration as soon as it finds a solution better than the current solution. The candidate solution search is more comprehensive when you use parallel computing. Although the number of iterations might be larger, the optimization without using parallel computing might be faster.

  To learn more about the expected speedup, see "Parallel Computing with the Pattern search Method" on page 2-61.

Why Doesn't the Estimation Using Parallel Computing Make Any Progress?

In some cases, the gradient computations on the remote worker machines may silently error out when you use parallel computing. In such cases, the **Estimation progress** table shows that the `f(x)` values do not change, and the optimization terminates after two iterations.

To troubleshoot the problem:

1 Run the optimization for a few iterations without parallel computing to see if the optimization progresses.
2 Check whether the remote workers have access to all model dependencies. Model dependencies include data variables and files required by the model to run.

   To learn more, see "Model Dependencies" on page 2-63.

Why Does the Estimation Using Parallel Computing Continue When I Click Stop?

When you use parallel computing with the `Pattern search` method, the software must wait until the current optimization iteration completes before it notifies the workers to stop. The optimization does not terminate immediately when you click **Stop**, and, instead, appears to continue running.

See Also
sdo.OptimizeOptions | parpool | sdo.getModelDependencies | sdo.optimize

More About
- "Speedup Parameter Estimation Using Parallel Computing" on page 2-59

Estimating Initial Conditions for Blocks with External Initial Conditions

When an integrator block uses an initial-condition port, which you specify by an IC block, you cannot estimate the initial conditions (ICs) of the integrator using Simulink Design Optimization software. Estimation is not possible because external ICs have priority over the ICs of a specific block to maintain the integrity of the model.

To tune the ICs of an integrator block with external ICs, you must modify the model to make the external signal into a tunable parameter. For example, you can set the IC block that feeds into the integrator to be a tunable variable and estimate it.

Estimation Sessions

In this section...
"Structure of an Estimation Session" on page 2-71
"Save Parameter Estimation Tool Sessions" on page 2-71
"Load Parameter Estimation Tool Sessions" on page 2-72
"Load Legacy Projects" on page 2-72

Structure of an Estimation Session

The Parameter Estimation tool, which is a tool for performing parameter estimation and validation, stores and organizes data from a given Simulink model inside a *session*. To open the Parameter Estimation tool, select **Analysis** > **Parameter Estimation** in the Simulink model window.

When using the Parameter Estimation tool, you can:

- Manage estimation sessions
- Select parameters and initial conditions to configure the estimation
- Specify cost functions
- Import signal data (to be matched by the input and output of your Simulink model)
- Specify the initial conditions of your model

Each estimation session can include:

- One or more estimation or validation experiments
- Parameter information
- Different settings or configurations for each experiment

The default session name is the same as the Simulink model name. The session name is shown on the title pane of Parameter Estimation tool.

Save Parameter Estimation Tool Sessions

To save your session as a MAT-file, click **Save Session** drop down list on the **Parameter Estimation** tab.
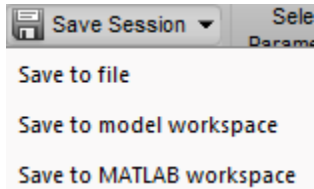
You have the options to browse to the location where you want to save the session using the `Save to file` option, or save to model workspace or MATLAB workspace.

Load Parameter Estimation Tool Sessions

To open previously saved sessions, click the **Open Session** drop-down menu.
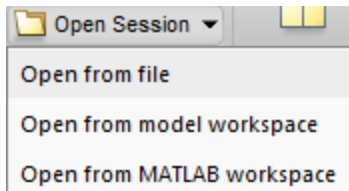
You have the option to browse to where the session file is, or open a session from model workspace or the MATLAB workspace. All sessions are MAT-files.

Load Legacy Projects

You can open legacy projects that are in MAT-files by selecting `Open from file` from the **Open Session** drop-down list. The Parameter Estimation tool recognizes and converts them into the new session format.

How the Software Formulates Parameter Estimation as an Optimization Problem

In this section...
"Overview of Parameter Estimation as an Optimization Problem" on page 2-73
"Cost Function" on page 2-73
"Bounds and Constraints" on page 2-75
"Optimization Methods and Problem Formulations" on page 2-76

Overview of Parameter Estimation as an Optimization Problem

When you perform parameter estimation, the software formulates an optimization problem. The optimization problem solution is the estimated parameter values set. This optimization problem consists of:

- x — *Design variables*. The model parameters and initial states to be estimated.
- $F(x)$ — *Objective function*. A function that calculates a measure of the difference between the simulated and measured responses. Also called *cost function* or *estimation error*.
- (Optional) $\underline{x} \le x \le \bar{x}$ — *Bounds*. Limits on the estimated parameter values.
- (Optional) $C(x)$ — *Constraint function*. A function that specifies restrictions on the design variables.

The optimization solver tunes the values of the design variables to satisfy the specified objectives and constraints. The exact formulation of the optimization depends on the optimization method that you use.

Cost Function

- "Types" on page 2-74
- "Time Base" on page 2-74

The software tunes the model parameters to obtain a simulated response (y_{sim}) that tracks the measured response or reference signal (y_{ref}). To do so, the solver minimizes the *cost function* or *estimation error*, a measure of the difference between the simulated and measured responses. The cost function, $F(x)$, is the objective function of the optimization problem.

Types

The raw estimation error, $e(t)$, is defined as:

$$e(t) = y_{ref}(t) - y_{sim}(t)$$

$e(t)$ is also referred to as the *error residuals* or, simply, *residuals*.

Simulink Design Optimization software provides you the following cost functions to process $e(t)$:

Cost Function	Formulation	Option Name in GUI or Command Line
Sum squared error (default)	$$F(x) = \sum_{t=0}^{t_N} e(t) \times e(t)$$ N is the number of samples.	`'SSE'`
Sum absolute error	$$F(x) = \sum_{t=0}^{t_N} |e(t)|$$ N is the number of samples.	`'SAE'`
Raw error	$$F(x) = \begin{bmatrix} e(0) \\ \vdots \\ e(N) \end{bmatrix}$$ N is the number of samples.	`'Residuals'` This option is available only at the command line.
Custom function	N/A	This option is available only at the command line.

Time Base

The software evaluates the cost function for a specific time interval. This interval is dependent on the *measured signal time base* and the *simulated signal time base.*

- The measured signal time base consists of all the time points for which the measured signal is specified. In case of multiple measured signals, this time base is the union of the time points of all the measured signals.

- The simulated signal time base consists of all the time points for which the model is simulated.

If the model uses a variable-step solver, then the simulated signal time base can change from one optimization iteration to another. The simulated and measured signal time bases can be different. The software evaluates the cost function for only the time interval that is common to both. By default, the software uses only the time points specified by the measured signal in the common time interval.

- In the GUI, you can specify the simulation start and stop times in the **Simulation time** area of the **Simulation Options** dialog box.
- At the command line, the software specifies the simulation stop time as the last point of the measured signal time base. For example, the following code simulates the model until the end time of the longest running output signal of `exp`, an `sdo.Experiment` object:

```
sim_obj = createSimulator(exp);
sim_obj = sim(sim_obj);
```

`sim_obj` contains the simulated response for the model associated with `exp`.

Bounds and Constraints

You can specify bounds for the design variables (estimated model parameters), based on your knowledge of the system. Bounds are expressed as:

$$\underline{x} \leq x \leq \bar{x}$$

$\underline{x}$ and $\bar{x}$ are the lower and upper bounds for the design variables.

For example, in a battery discharging experiment, the estimated battery initial charge must be greater than zero and less than `Inf`. These bounds are expressed as:

$$0 < x < \infty$$

For an example of how to specify these types of bounds, see "Estimate Model Parameters and Initial States (Code)" on page 2-103.

You can also specify other constraints, $C(x)$, on the design variables at the command line. $C(x)$ can be linear or nonlinear and can describe equalities or inequalities. $C(x)$ can also specify multiparameter constraints. For example, for a simple friction model, $C(x)$ can

specify that the static friction coefficient must be greater than or equal to the dynamic friction coefficient. One way of expressing this constraint is:

$$C(x) : x_1 - x_2$$
$$C(x) \le 0$$

x_1 and x_2 are the dynamic and static friction coefficients, respectively.

For an example of how to specify a constraint, see "Estimate Model Parameters with Parameter Constraints (Code)" on page 2-140.

Optimization Methods and Problem Formulations

An optimization problem can be one of the following types:

- Minimization problem — Minimizes an objective function, $F(x)$. You specify the measured signal that you want the model output to track. You can optionally specify bounds for the estimated parameters.

- Mixed minimization and feasibility problem — Minimizes an objective function, $F(x)$, subject to specified bounds and constraints, $C(x)$. You specify the measured signal that you want the model to track and bounds and constraints for the estimated parameters.

- Feasibility problem — Finds a solution that satisfies the specified constraints, $C(x)$. You specify only bounds and constraints for the estimated parameters. This type of problem is not common in parameter estimation.

The optimization method that you specify determines the formulation of the estimation problem. The software provides the following optimization methods:

Optimization Method Name	Description	Optimization Problem Formulation
• User interface: **Nonlinear Least Squares** • Command line: `'lsqnonlin'`	Minimizes the squares of the residuals, recommended method for parameter estimation. This method requires a vector of error	**Minimization Problem** $$min_{x} \|F(x)\|_2^2 = min_{x}\left(f_1(x)^2 + f_2(x)^2 + \ldots + f_n(x)^2 \right)$$ $$s.t. \ \underline{x} \le x \le \overline{x}$$ $f_1(x)$, $f_2(x)$,...,$f_n(x)$ represent residuals. n is the number of samples.

Optimization Method Name	Description	Optimization Problem Formulation
	residuals, computed using a fixed time base. Do not use this approach if you have a scalar cost function or if the number of error residuals can change from one iteration to another. This method uses the Optimization Toolbox function, `lsqnonlin`.	**Mixed Minimization and Feasibility Problem** Not supported. **Feasibility Problem** Not supported.

Optimization Method Name	Description	Optimization Problem Formulation
• User interface: **Gradient Descent** • Command line: `'fmincon'`	General nonlinear solver, uses the cost function gradient. Use this approach if you want to specify one or any combination of the following: • Custom cost functions • Parameter-based constraints • Signal-based constraints This method uses the Optimization Toolbox function, `fmincon`. For information on how the gradient is computed, see "Gradient Computations" on page 2-90.	**Minimization Problem** $$\min_x F(x)$$ $$s.t. \ \underline{x} \le x \le \bar{x}$$ **Mixed Minimization and Feasibility Problem** $$\min_x \ F(x)$$ $$s.t. \quad C(x) \le 0$$ $$\underline{x} \le x \le \bar{x}$$ --- **Note:** When tracking a reference signal, the software ignores the maximally feasible solution option. --- **Feasibility Problem** • If you select the maximally feasible solution option (i.e., the optimization continues after an initial feasible solution is found), the software uses the following problem formulation: $$\min_{[x,\gamma]} \ \gamma$$ $$s.t. \quad C(x) \le \gamma$$ $$\underline{x} \le x \le \bar{x}$$ $$\gamma \le 0$$ γ is a slack variable that permits a feasible solution with $C(x) \le \gamma$ rather than $C(x) \le 0$.

Optimization Method Name	Description	Optimization Problem Formulation
		• If you do not select the maximally feasible solution option (i.e., the optimization terminates as soon as a feasible solution is found), the software uses the following problem formulation: $$\min_{x} \; 0$$ $$s.t. \;\; C(x) \le 0$$ $$\underline{x} \le x \le \overline{x}$$

Optimization Method Name	Description	Optimization Problem Formulation
• User interface: **Simplex Search** • Command line: `'fminsearch'`	Based on the Nelder-Mead algorithm, this approach does not use the cost function gradient. Use this approach if your cost function or constraints are not continuous or differentiable. This method uses the Optimization Toolbox functions, `fminsearch` and `fminbnd`. `fminbnd` is used if one scalar parameter is being optimized. Otherwise, `fminsearch` is used. You cannot specify parameter bounds, $\underline{x} \le x \le \overline{x}$, with `fminsearch`.	**Minimization Problem** $$\min_x F(x)$$ **Mixed Minimization and Feasibility Problem** The software formulates the problem in two steps: 1 Finds a feasible solution. $$\min_x \max(C(x))$$ 2 Minimizes the objective. The software uses the results from step 1 as initial guesses. It maintains feasibility by introducing a discontinuous barrier in the optimization objective. $$\min_x \Gamma(x)$$ where $$\Gamma(x) = \begin{cases} \infty & \text{if } \max(C(x)) > 0 \\ F(x) & \text{otherwise.} \end{cases}$$ **Feasibility Problem** $$\min_x \max(C(x))$$

Optimization Method Name	Description	Optimization Problem Formulation
• User interface: **Pattern Search** • Command line: `'patternsearch'`	Direct search method, based on the generalized pattern search algorithm, this method does not use the cost function gradient. Use this approach if your cost function or constraints are not continuous or differentiable. This method uses the Global Optimization Toolbox function, `patternsearch`.	**Minimization Problem** $$\min_{x} \quad F(x)$$ $$s.t. \quad \underline{x} \leq x \leq \overline{x}$$ **Mixed Minimization and Feasibility Problem** The software formulates the problem in two steps: 1 Finds a feasible solution. $$\min_{x} \max(C(x))$$ $$s.t. \quad \underline{x} \leq x \leq \overline{x}$$ 2 Minimizes the objective. The software uses the results from step 1 as initial guesses. It maintains feasibility by introducing a discontinuous barrier in the optimization objective. $$\min_{x} \Gamma(x)$$ $$s.t. \ \underline{x} \leq x \leq \overline{x}$$ where $$\Gamma(x) = \begin{cases} \infty & \text{if } \max(C(x)) > 0 \\ F(x) & \text{otherwise.} \end{cases}$$ **Feasibility Problem** $$\min_{x} \max(C(x))$$ $$s.t. \quad \underline{x} \leq x \leq \overline{x}$$

See Also

```
fminbnd | fmincon | fminsearch | lsqnonlin | patternsearch
| sdo.Experiment | sdo.requirements.SignalTracking |
sdo.requirements.SignalTracking | sdo.SimulationTest
```

Related Examples

- "Estimate Model Parameter Values (Code)" on page 2-91
- "Estimate Model Parameters with Parameter Constraints (Code)" on page 2-140
- "Estimate Parameters from Measured Data"

More About

- "Writing a Cost Function" on page 2-83

Writing a Cost Function

Cost Function Overview

When you use `sdo.optimize` to optimize model parameters (*design variables*), you must provide a MATLAB function as an input to `sdo.optimize`. This function, also called a *cost function*, must evaluate the cost and constraint values for the design variable values for an iteration. (The cost and constraint functions are collectively referred to as *requirements*.) `sdo.optimize` calls this function at every optimization iteration and use the function output to decide the optimization direction.

The cost function can also be used for global sensitivity analysis. You generate samples of the model parameters and evaluate the cost function for each sample using `sdo.evaluate`.

The cost function must have:

- Input — `params`, a vector of the design variables (`param.Continuous` objects) to be optimized.
- Output:

    - (Required) `vals`, a structure with one or more fields that specify the values of the cost and constraint violations.
    - (Optional) `derivs`, a structure with one or more fields that specify the values of the gradients of the cost and constraint violations.

You perform the following tasks within the function:

- Extract the current design variable values from `params`.
- If the simulated response is required for evaluating the requirements, then simulate the model using the current design variable values.

- Evaluate the requirements.
- Specify the requirement values as fields of `vals`.

To use a cost function with `sdo.optimize`, enter:

```
[param_opt,opt_info] = sdo.optimize(@myCostFunc,param)
```

Here, `myCostFunc` is the name of the MATLAB function and `param` is a vector of the design variables.

Similarly, to use a cost function with `sdo.evaluate`, enter:

```
[y,info] = sdo.evaluate(@myCostFunc,param)
```

Convenience Objects

The software provides you with the following convenience objects that can you can use in the cost function:

Class Name	Description
`sdo.SimulationTest`	Use an `sdo.SimulationTest` object, also referred to as a *simulator*, to simulate a model. The simulator allows you to simulate the model using alternative inputs, model parameter and initial-state values, without modifying the model. You configure the simulator to log the signals needed to evaluate requirements and use the `sim` method to simulate the model. Then, you extract the model response from the object and evaluate the requirements.
Requirements objects: Time-domain requirements: • `sdo.requirements.SignalBound` • `sdo.requirements.StepResponseEnvelop` • `sdo.requirements.SignalTracking`	Use these *requirements objects* to specify time- and frequency-domain costs or constraints on the design variables. You configure the properties of the object and then use the object's `evalRequirement` method to evaluate how closely the current design variables satisfy your design requirement.

Class Name	Description
Frequency-domain requirements: • `sdo.requirements.GainPhaseMargin` • `sdo.requirements.BodeMagnitude` • `sdo.requirements.ClosedLoopPeakGain` • `sdo.requirements.PZDampingRatio` • `sdo.requirements.PZNaturalFrequency` • `sdo.requirements.PZSettlingTime` • `sdo.requirements.SignalTracking` • `sdo.requirements.StepResponseEnvelope` • `sdo.requirements.OpenLoopGainPhase`	
`sdo.Experiment`	Use an `sdo.Experiment` object, also referred to as simply an *experiment*, to specify the input/output data, model parameter and initial-state values for parameter estimation. You update the design variable values associated with the experiment using the `setEstimatedValues` method. Then, you create a simulator, using the `createSimulator` method, to simulate the model using the updated model configuration.

Inputs

- "Model Parameters and States" on page 2-85
- "Multiple Inputs" on page 2-86

Model Parameters and States

The function must take as input a vector of model parameter objects (`param.Continuous` objects) and, optionally, initial-state objects (`param.State` objects). These objects represent the *design variables* of the optimization problem. You obtain these objects by using the `sdo.getParameterFromModel` and `sdo.getStateFromModel` commands.

To access a design variable value, use:

```
param_val = p(1).Value;
```

Here, `p` is a vector of `param.Continuous` objects and `p(1)` is either a model parameter or an initial-state object.

Multiple Inputs

`sdo.optimize` requires that the cost function accept only one input argument, `params`. However, you might want to use additional inputs. For instance, you could make the model name an input argument and configure the function to be used for multiple models. To call `sdo.optimize` and use a function that accepts more than one input argument, you use an anonymous function. For example, suppose `myCostFunc_mult_inputs` is a cost function that takes `param`, `arg1`, and `arg2` as inputs. Then, assuming that all input arguments are variables in the workspace, you enter:

```
myCostFunc = @(param) myCostFunc_mult_inputs(param,arg1,arg2);
[param_opt,opt_info] = sdo.optimize(@myCostFunc,param);
```

Additional inputs can also help reduce code redundancy and computation cost, given that the function is called repeatedly by `sdo.optimize` during optimization. For instance, if you use a convenience object in your function, you can create it once, before calling `sdo.optimize`. Then, you can modify the convenience object's properties as required within the function for each iteration.

Evaluate Requirements

The core of the function is where you evaluate how well the current design variables satisfy the design requirements. You can use MATLAB functions to do so. You can also use the requirements objects that the Simulink Design Optimization software provides. These objects enable you to specify requirements such as step-response characteristics, gain/phase margin bounds, Bode magnitude bounds, etc.

- Parameter-only requirements — Extract the design variable values and compute the requirement values.

  For example, you can minimize the cylinder cross-sectional area, a design variable, in a hydraulic cylinder. See "Design Optimization to Meet a Custom Objective (Code)".

- Model response-based requirements — Simulate the model using the current design variable values, extract the model response, and compute the requirement values.

There are multiple ways to simulate the model, including:

- Using an `sdo.SimulationTest` object. You update the model parameter values using the simulator's `Parameters` property. Then, you use the `sim` method to simulate the model and extract the logged signals from the simulator that are of interest. For an example, see "Design Optimization to Meet a Custom Objective (Code)".

  In parameter estimation, you can use the `createSimulator` method of the `sdo.Experiment` object to create the simulator. Before creating the simulator, you update the experiment with the current design variable values using the `setEstimatedValues` method. For an example, see "Estimate Model Parameters Per Experiment (Code)" on page 2-127

- Using `sdo.setValueInModel` to update the model and then calling `sim` to simulate the model.

- Linear model-based requirements — Update the model with the current design variable values, linearize the model, and compute the requirement values.

  Use `sdo.setValueInModel` to update the model and functions such as `linmod` or `linearize` to linearize the model. `linearize` requires a Simulink Control Design™ license.

Outputs

- "Cost and Constraint Values" on page 2-87
- "Multiple Objectives" on page 2-88

Cost and Constraint Values

Your function must return a structure containing the cost and constraint values for the current design variables. This structure must have one or more of the following fields, as required by your optimization problem:

- `F` — Cost value.
- `Cleq`, `Ceq` — Nonlinear constraint values. The solver satisfies $Cleq \leq 0$ and $Ceq = 0$.
- `leq`, `eq` — Linear constraint values. The solver satisfies $leq \leq 0$ and $eq = 0$.

If you have multiple constraints of one type, concatenate the values into a vector, and specify this vector as the corresponding field value. For instance, if you have a

hydraulic cylinder, you can specify nonlinear inequality constraints on the piston position (`Cleq1`) and cylinder pressure (`Cleq2`). In this case, specify the `Cleq` field of the output structure, `vals`, as:

```
vals.Cleq = [Cleq1; Cleq2];
```

For an example, see "Design Optimization to Meet a Custom Objective (Code)".

By default, the software computes the cost and constraint gradients using numeric perturbation. However, you can specify the gradients and return them as an additional output. This output must be a structure with one or more of the following fields, as required by your optimization problem:

- `F` — Cost derivatives.
- `Cleq` — Nonlinear inequality constraints derivatives.
- `Ceq` — Nonlinear equality constraints derivatives.

You must also set the `GradFcn` property of the optimization option set to `'on'`.

Multiple Objectives

Simulink Design Optimization does not support multi-objective optimization. However, you can return the cost value (`F`) as a vector, representing the multiple objective values. Using this approach does not halt the optimization. Instead, the software sums the elements of the vector and minimizes this sum. The exception to this behavior is if you are using the nonlinear least squares (`lsqnonlin`) optimization method. The nonlinear least squares method, used for parameter estimation, requires that you return the error residuals as a vector. In this case, the software minimizes the sum square of this vector.

If you are tracking multiple signals and using `lsqnonlin`, then you must concatenate the error residuals for the different signals into one vector. Specify this vector as the `F` field value.

For an example of single objective optimization using the gradient descent method, see "Design Optimization to Meet a Custom Objective (Code)".

For an example of multiple objective optimization using the nonlinear least squares method, see "Estimate Model Parameters Per Experiment (Code)" on page 2-127.

See Also
sdo.OptimizeOptions | param.Continuous | sdo.SimulationTest | sdo.Experiment | sdo.optimize | sdo.setValueInModel

Related Examples

- "Design Optimization to Meet a Custom Objective (Code)"
- "Estimate Model Parameter Values (Code)" on page 2-91

More About

- "How the Optimization Algorithm Formulates Minimization Problems"
- "How the Software Formulates Parameter Estimation as an Optimization Problem" on page 2-73

Gradient Computations

For the `Gradient descent (fmincon)` optimization solver, the gradients are computed using numerical perturbation:

$$dx = \sqrt[3]{eps} \times \max\left(|x|, \frac{1}{10}x_{typical}\right)$$

$$dL = \max(x - dx, x_{\min})$$

$$dR = \min(x + dx, x_{\max})$$

$$F_L = opt\_fcn(dL)$$

$$F_R = opt\_fcn(dR)$$

$$\frac{dF}{dx} = \frac{(F_L - F_R)}{(dL - dR)}$$

- x is a scalar design variable.
- x_{min} is the lower bound of x.
- x_{max} is the upper bound of x.
- $x_{typical}$ is the scaled value of x.
- opt_fcn is the objective function.

dx is relatively large to accommodate simulation solver tolerances.

If you want to compute the gradients in any other way, you can do so in the cost function you write for performing design optimization programmatically. See `sdo.optimize` and GradFcn of `sdo.OptimizeOptions` for more information.

See Also
`fmincon`

More About
- "How the Software Formulates Parameter Estimation as an Optimization Problem" on page 2-73
- "How the Optimization Algorithm Formulates Minimization Problems"

Estimate Model Parameter Values (Code)

This example shows how to use experimental data to estimate model parameter values.

Aircraft Model

The Simulink model, `sdoAircraftEstimation`, models the longitudinal flight control system of an aircraft.

```
open_system('sdoAircraftEstimation')
```

Aircraft Longitudinal Flight Control

This demonstration models a flight control algorithm
of an aircraft.

Copyright 1990-2012 The MathWorks, Inc.
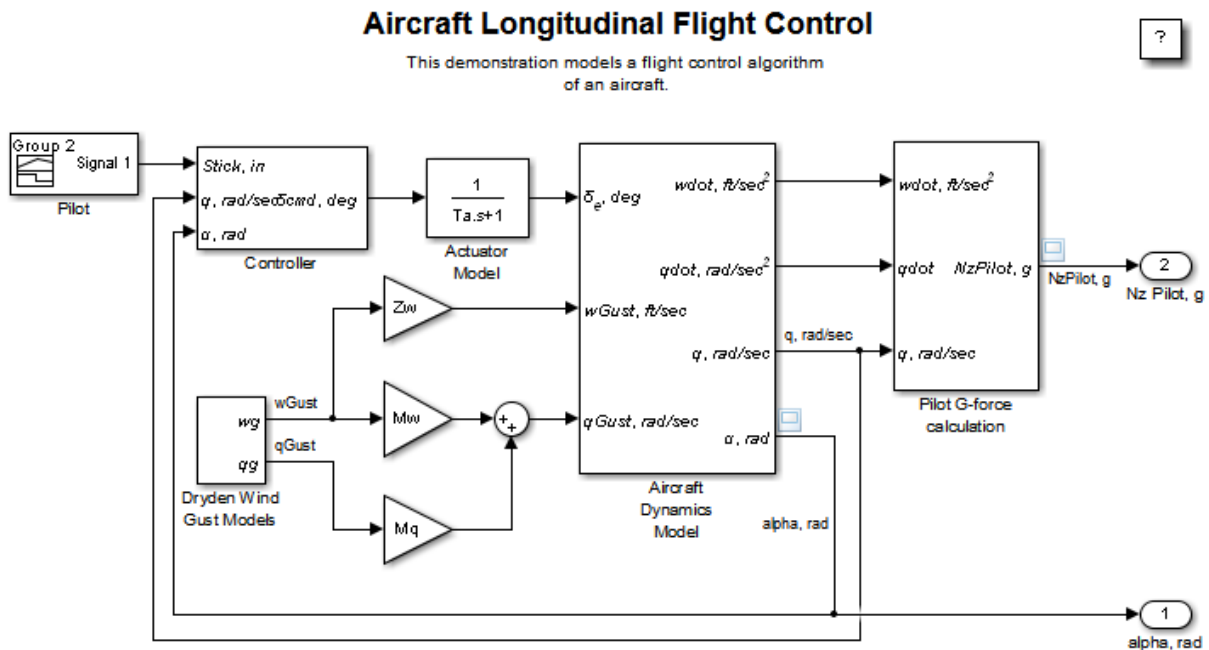
Estimation Problem

You use measured data to estimate the aircraft model parameters and states.

Measured output data:

- Pilot G force, output of the `Pilot G-force calculation` block
- Angle of attack, fourth output of the `Aircraft Dynamics Model` block

Parameters:

- Actuator time constant, `Ta`, used by the `Actuator Model` block
- Vertical velocity, `Zd`, used by the `Aircraft Dynamics Model` block
- Pitch rate gains, `Md`, used by the `Aircraft Dynamics Model` block

State:

- Initial state of the first-order actuator model, `sdoAircraftEstimation/Actuator Model`

Define the Estimation Experiment

Get the measured data.

```
[time,iodata] = sdoAircraftEstimation_Experiment;
```

The `sdoAircraftEstimation_Experiment` function returns the measured output data, `iodata`, and the corresponding time vector. The first column of `iodata` is the pilot G force and the second column is the angle of attack.

To see the code for this function, type `edit sdoAircraftEstimation_Experiment`.

Create an experiment object to store the measured input/output data.

```
Exp = sdo.Experiment('sdoAircraftEstimation');
```

Create an object to store the measured pilot G-Force output.

```
PilotG = Simulink.SimulationData.Signal;
PilotG.Name      = 'PilotG';
PilotG.BlockPath = 'sdoAircraftEstimation/Pilot G-force calculation';
PilotG.PortType  = 'outport';
PilotG.PortIndex = 1;
PilotG.Values    = timeseries(iodata(:,2),time);
```

Create an object to store the measured angle of attack (alpha) output.

```
AoA           = Simulink.SimulationData.Signal;
AoA.Name      = 'AngleOfAttack';
AoA.BlockPath = 'sdoAircraftEstimation/Aircraft Dynamics Model';
AoA.PortType  = 'outport';
AoA.PortIndex = 4;
AoA.Values    = timeseries(iodata(:,1),time);
```

Add the measured pilot G-Force and angle of attack data to the experiment as the expected output data.

```
Exp.OutputData = [...
    PilotG; ...
    AoA];
```

Add the initial state for the `Actuator Model` block to the experiment. Set its `Free` field to `true` so that it is estimated.

```
Exp.InitialStates = sdo.getStateFromModel('sdoAircraftEstimation','Actuator Model');
Exp.InitialStates.Minimum = 0;
Exp.InitialStates.Free    = true;
```

Compare the Measured Output and the Initial Simulated Output

Create a simulation scenario using the experiment and obtain the simulated output.

```
Simulator     = createSimulator(Exp);
Simulator     = sim(Simulator);
```

Search for the pilot G-Force and angle of attack signals in the logged simulation data.

```
SimLog       = find(Simulator.LoggedData,get_param('sdoAircraftEstimation','SignalLogg
PilotGSignal = find(SimLog,'PilotG');
AoASignal    = find(SimLog,'AngleOfAttack');
```

Plot the measured and simulated data.

As expected, the model response does not match the experimental output data.

```
plot(time, iodata, ...
    AoASignal.Values.Time,AoASignal.Values.Data,'--', ...
    PilotGSignal.Values.Time,PilotGSignal.Values.Data,'-.');
title('Simulated and Measured Responses Before Estimation')
legend('Measured angle of attack',  'Measured pilot g force', ...
     'Simulated angle of attack', 'Simulated pilot g force');
```

Simulated and Measured Responses Before Estimation
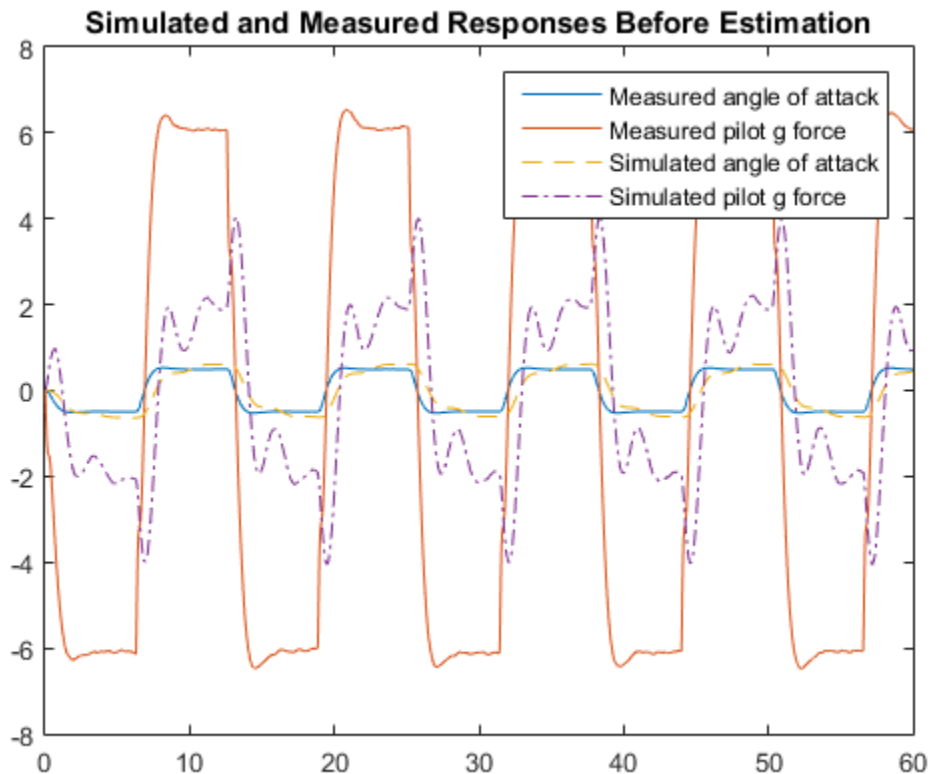
Specify the Parameters to Estimate

Select the model parameters that describe the flight control actuation system. Specify bounds for the estimated parameter values based on our understanding of the actuation system.

```
p = sdo.getParameterFromModel('sdoAircraftEstimation',{'Ta','Md','Zd'});
p(1).Minimum = 0.01;   %Ta
p(1).Maximum = 1;
p(2).Minimum = -10;    %Md
p(2).Maximum = 0;
p(3).Minimum = -100;   %Zd
p(3).Maximum = 0;
```

Get the actuator initial state value that is to be estimated from the experiment.

```
s = getValuesToEstimate(Exp);
```

Group the model parameters and initial states to be estimated together.

```
v = [p;s]
```

```
v(1,1) =

      Name: 'Ta'
     Value: 0.5000
   Minimum: 0.0100
   Maximum: 1
      Free: 1
     Scale: 0.5000
      Info: [1x1 struct]


v(2,1) =

      Name: 'Md'
     Value: -1
   Minimum: -10
   Maximum: 0
      Free: 1
     Scale: 1
      Info: [1x1 struct]


v(3,1) =

      Name: 'Zd'
     Value: -80
   Minimum: -100
   Maximum: 0
      Free: 1
     Scale: 128
      Info: [1x1 struct]


v(4,1) =

      Name: 'sdoAircraftEstimation/Actuator
```

```
Model'
      Value: 0
    Minimum: 0
    Maximum: Inf
       Free: 1
      Scale: 1
    dxValue: 0
     dxFree: 1
       Info: [1x1 struct]


4x1 param.Continuous
```

Define the Estimation Objective Function

Create an estimation objective function to evaluate how closely the simulation output, generated using the estimated parameter values, matches the measured data.

Use an anonymous function with one input argument that calls the sdoAircraftEstimation_Objective function. We pass the anonymous function to sdo.optimize, which evaluates the function at each optimization iteration.

```
estFcn = @(v) sdoAircraftEstimation_Objective(v,Exp);
```

The sdoAircraftEstimation_Objective function:

- Has one input argument that specifies the actuator parameter values and the actuator initial state.

- Has one input argument that specifies the experiment object containing the measured data.

- Returns a vector of errors between simulated and experimental outputs.

The sdoAircraftEstimation_Objective function requires two inputs, but sdo.optimize requires a function with one input argument. To work around this, estFcn is an anonymous function with one input argument, v, but it calls sdoAircraftEstimation_Objective using two input arguments, v and Exp.

For more information regarding anonymous functions, see "Anonymous Functions".

The sdo.optimize command minimizes the return argument of the anonymous function estFcn, that is, the residual errors returned by sdoAircraftEstimation_Objective. For more details on how to write an

objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

To examine the estimation objective function in more detail, type `edit sdoAircraftEstimation_Objective` at the MATLAB command prompt.

type sdoAircraftEstimation_Objective

```
function vals = sdoAircraftEstimation_Objective(v,Exp)
%SDOAIRCRAFTESTIMATION_OBJECTIVE
%
%    The sdoAircraftEstimation_Objective function is used to compare model
%    outputs against experimental data.
%
%    vals = sdoAircraftEstimation_Objective(v,Exp)
%
%    The |v| input argument is a vector of estimated model parameter values
%    and initial states.
%
%    The |Exp| input argument contains the estimation experiment data.
%
%    The |vals| return argument contains information about how well the
%    model simulation results match the experimental data and is used by
%    the |sdo.optimize| function to estimate the model parameters.
%
%    See also sdo.optimize, sdoExampleCostFunction,
%    sdoAircraftEstimation_cmddemo
%

% Copyright 2012 The MathWorks, Inc.

%%
% Define a signal tracking requirement to compute how well the model output
% matches the experiment data. Configure the tracking requirement so that
% it returns the tracking error residuals (rather than the
% sum-squared-error) and does not normalize the errors.
%
r = sdo.requirements.SignalTracking;
r.Type      = '==';
r.Method    = 'Residuals';
r.Normalize = 'off';

%%
% Update the experiments with the estimated parameter values.
```

```
%
Exp  = setEstimatedValues(Exp,v);

%%
% Simulate the model and compare model outputs with measured experiment
% data.
%
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);

SimLog      = find(Simulator.LoggedData,get_param('sdoAircraftEstimation','SignalLogg:
PilotGSignal = find(SimLog,'PilotG');
AoASignal    = find(SimLog,'AngleOfAttack');

PilotGError = evalRequirement(r,PilotGSignal.Values,Exp.OutputData(1).Values);
AoAError     = evalRequirement(r,AoASignal.Values,Exp.OutputData(2).Values);

%%
% Return the residual errors to the optimization solver.
%
vals.F = [PilotGError(:); AoAError(:)];
end
```

Estimate the Parameters

Use the `sdo.optimize` function to estimate the actuator parameter values and initial state.

Specify the optimization options. The estimation function `sdoAircraftEstimation_Objective` returns the error residuals between simulated and experimental data and does not include any constraints, making this problem ideal for the 'lsqnonlin' solver.

```
opt = sdo.OptimizeOptions;
opt.Method = 'lsqnonlin';
```

Estimate the parameters.

```
vOpt = sdo.optimize(estFcn,v,opt)


 Optimization started 20-Jan-2015 08:40:58

                          Step-size First-order
 Iter F-count        f(x)           optimality
```

```
 0      8       27972.2              1
 1     17       10124.8         0.4744         5.69e+04
 2     26       3127.92         0.3854         1.24e+04
 3     35       872.751         0.4286         2.81e+03
 4     44        238.66         0.5147              618
 5     53       71.9182          0.493              147
 6     62       17.2194         0.4163             44.9
 7     71       1.82827         0.3069             11.4
 8     80     0.0440753         0.1321             1.38
 9     89     0.0020167        0.03129           0.0871
10     98   0.000244535        0.008145           0.119
11    107   5.74974e-05        0.005686          0.00522
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

vOpt(1,1) =

      Name: 'Ta'
     Value: 0.0500
   Minimum: 0.0100
   Maximum: 1
      Free: 1
     Scale: 0.5000
      Info: [1x1 struct]


vOpt(2,1) =

      Name: 'Md'
     Value: -6.8848
   Minimum: -10
   Maximum: 0
      Free: 1
     Scale: 1
      Info: [1x1 struct]


vOpt(3,1) =

      Name: 'Zd'
     Value: -63.9983
   Minimum: -100
```

```
      Maximum: 0
         Free: 1
        Scale: 128
         Info: [1x1 struct]


vOpt(4,1) =

         Name: 'sdoAircraftEstimation/Actuator
Model'
        Value: 6.3839e-05
      Minimum: 0
      Maximum: Inf
         Free: 1
        Scale: 1
      dxValue: 0
       dxFree: 1
         Info: [1x1 struct]


4x1 param.Continuous
```

Compare the Measured Output and the Final Simulated Output

Update the experiments with the estimated parameter values.

```
Exp = setEstimatedValues(Exp,vOpt);
```

Simulate the model using the updated experiment and compare the simulated output with the experimental data.

The model response using the estimated parameter values closely matches the experiment output data.

```
Simulator    = createSimulator(Exp);
Simulator    = sim(Simulator);
SimLog       = find(Simulator.LoggedData,get_param('sdoAircraftEstimation','SignalLogg
PilotGSignal = find(SimLog,'PilotG');
AoASignal    = find(SimLog,'AngleOfAttack');

plot(time, iodata, ...
    AoASignal.Values.Time,AoASignal.Values.Data,'-.', ...
    PilotGSignal.Values.Time,PilotGSignal.Values.Data,'--')
title('Simulated and Measured Responses After Estimation')
```

```
legend('Measured angle of attack',  'Measured pilot g force', ...
    'Simulated angle of attack', 'Simulated pilot g force');
```
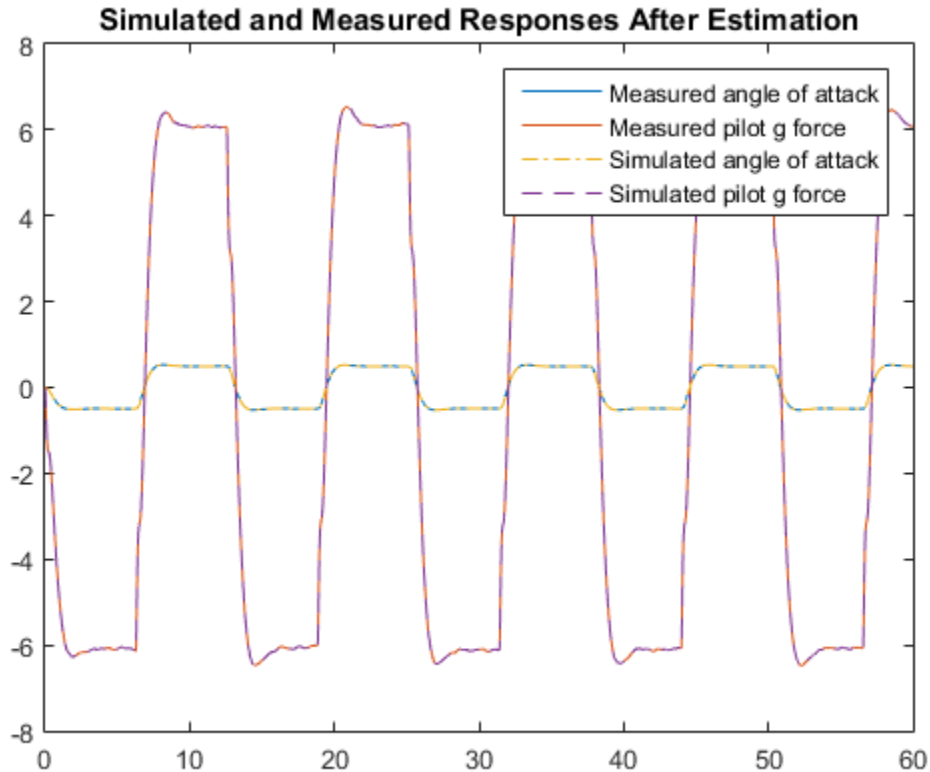

Update the Model Parameter Values

Update the model with the estimated actuator parameter values. Do not update the model actuator initial state (fourth element of vOpt) as it is dependent on the experiment.

```
sdo.setValueInModel('sdoAircraftEstimation',vOpt(1:3));
```

Related Examples

To learn how to estimate model parameters using the Parameter Estimation Tool, see "Estimate Model Parameter Values (GUI)".

2-101

Close the model

```
bdclose('sdoAircraftEstimation')
```

Estimate Model Parameters and Initial States (Code)

This example shows how to estimate the initial state and parameters of a model.

This example requires Simscape®.

RC Circuit Model

The Simulink model, `sdoRCCircuit`, models a simple resistor-capacitor (RC) circuit.

```
open_system('sdoRCCircuit');
```
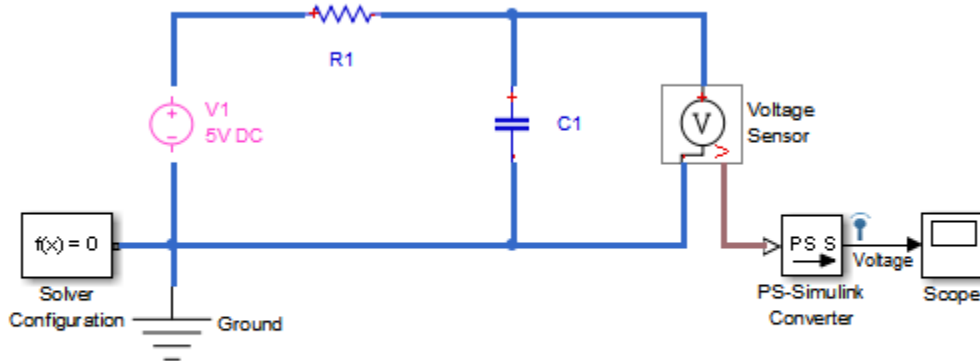

Copyright 2011 The MathWorks, Inc.

Estimation Problem

You use the measured data to estimate the RC model parameter and state values.

Measured output data:

· Capacitor voltage, output of the `PS-Simulink Converter` block

Parameter:

· Capacitance, `C1`, used by the `C1` block

State:

- Initial voltage of the capacitor, `C1`

Define the Estimation Experiment

Get the measured data.

```
load sdoRCCircuit_ExperimentData
```

The variables `time` and `data` are loaded into the workspace, where `data` is the measured capacitor voltage for times `time`.

Create an experiment object to store the experimental voltage data.

```
Exp = sdo.Experiment('sdoRCCircuit');
```

Create an object to store the measured capacitor voltage output.

```
Voltage = Simulink.SimulationData.Signal;
Voltage.Name      = 'Voltage';
Voltage.BlockPath = 'sdoRCCircuit/PS-Simulink Converter';
Voltage.PortType  = 'outport';
Voltage.PortIndex = 1;
Voltage.Values    = timeseries(data,time);
```

Add the measured capacitor data to the experiment as the expected output data.

```
Exp.OutputData = Voltage;
```

Compare the Measured Output and the Initial Simulated Output

Create a simulation scenario using the experiment and obtain the simulated output.

```
Simulator    = createSimulator(Exp);
Simulator    = sim(Simulator);
```

Search for the voltage signal in the logged simulation data.

```
SimLog     = find(Simulator.LoggedData,get_param('sdoRCCircuit','SignalLoggingName'))
Voltage    = find(SimLog,'Voltage');
```

Plot the measured and simulated data.

The model response does not match the experimental output data.

```
plot(time,data,'ro',Voltage.Values.Time,Voltage.Values.Data,'b')
title('Simulated and Measured Responses Before Estimation')
```

```
legend('Measured Voltage','Simulated Voltage')
```

Simulated and Measured Responses Before Estimation
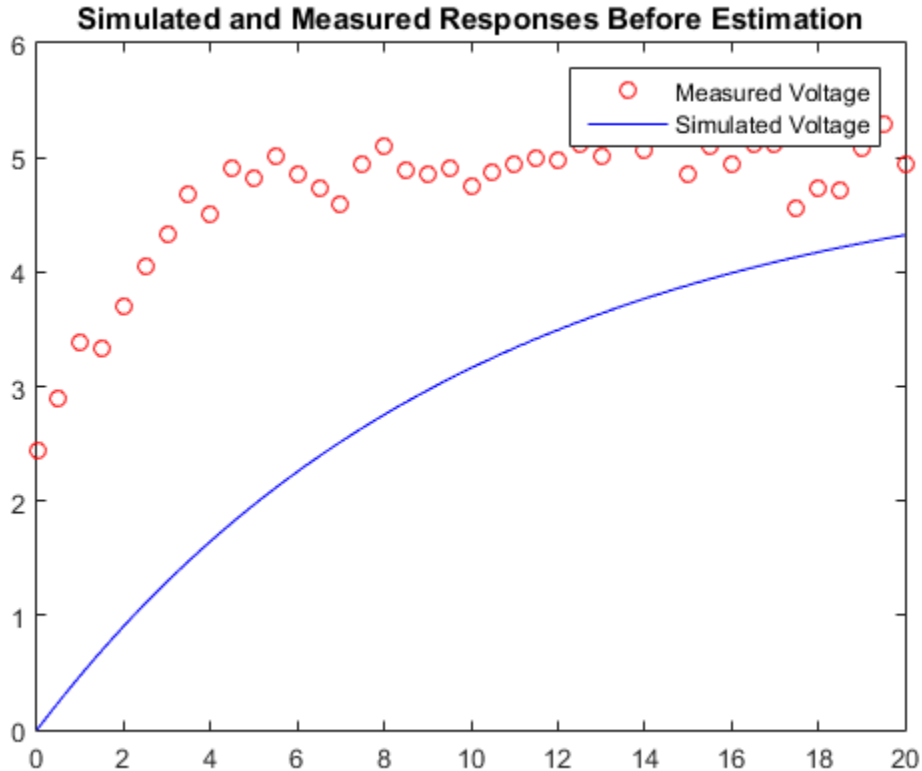
Specify the Parameters to Estimate

Select the capacitance parameter from the model. Specify an initial guess for the capacitance value (460 uF) and a minimum bound (0 F).

```
p = sdo.getParameterFromModel('sdoRCCircuit','C1');
p.Value   = 460e-6;
p.Minimum = 0;
```

Define the Estimation Objective Function

Create an estimation objective function to evaluate how closely the simulation output, generated using the estimated parameter value, matches the measured data.

Use an anonymous function with one input argument that calls the `sdoRCCircuit_Objective` function. We pass the anonymous function to `sdo.optimize`, which evaluates the function at each optimization iteration.

```
estFcn = @(v) sdoRCCircuit_Objective(v,Exp);
```

The `sdoRCCircuit_Objective` function:

- Has one input argument that specifies the estimated circuit capacitance value.

- Has one input argument that specifies the experiment object containing the measured data.

- Returns a vector of errors between simulated and experimental outputs.

The `sdoRCCircuit_Objective` function requires two inputs, but `sdo.optimize` requires a function with one input argument. To work around this, `estFcn` is an anonymous function with one input argument, `v`, but it calls `sdoRCCircuit_Objective` using two input arguments, `v` and `Exp`.

For more information regarding anonymous functions, see "Anonymous Functions".

The optimization solver minimizes the residual errors. For more details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

To examine the estimation object function in more detail, type `edit sdoRCCircuit_Objective` at the MATLAB command prompt.

```
type sdoRCCircuit_Objective
```

```
function vals = sdoRCCircuit_Objective(v,Exp)
%SDORCCIRCUIT_OBJECTIVE
%
%    The sdoRCCircuit_Objective function is used to compare model
%    outputs against experimental data.
%
%    vals = sdoRCCircuit_Objective(v,Exp)
%
%    The |v| input argument is a vector of estimated model parameter values
%    and initial states.
%
%    The |Exp| input argument contains the estimation experiment data.
%
```

```
%     The |vals| return argument contains information about how well the
%     model simulation results match the experimental data and is used by
%     the |sdo.optimize| function to estimate the model parameters.
%
%     See also sdo.optimize, sdoExampleCostFunction, sdoRCCircuit_cmddemo
%

% Copyright 2012 The MathWorks, Inc.

%%
% Define a signal tracking requirement to compute how well the model output
% matches the experiment data. Configure the tracking requirement so that
% it returns the tracking error residuals (rather than the
% sum-squared-error) and does not normalize the errors.
%
r = sdo.requirements.SignalTracking;
r.Type      = '==';
r.Method    = 'Residuals';
r.Normalize = 'off';

%%
% Update the experiments with the estimated parameter values.
%
Exp  = setEstimatedValues(Exp,v);

%%
% Simulate the model and compare model outputs with measured experiment
% data.
%
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);

SimLog  = find(Simulator.LoggedData,get_param('sdoRCCircuit','SignalLoggingName'));
Voltage = find(SimLog,'Voltage');

VoltageError = evalRequirement(r,Voltage.Values,Exp.OutputData(1).Values);

%%
% Return the residual errors to the optimization solver.
%
vals.F = VoltageError(:);
end
```

Estimate the Parameters

Use the `sdo.optimize` function to estimate the capacitance value.

Specify the optimization options. The estimation function `sdoRCCircuit_Objective` returns the error residuals between simulated and experimental data and does not include any constraints, making this problem ideal for the 'lsqnonlin' solver.

```
opt = sdo.OptimizeOptions;
opt.Method = 'lsqnonlin';
```

Estimate the parameters.

```
pOpt = sdo.optimize(estFcn,p,opt)
```

```
 Optimization started 20-Jan-2015 08:41:46

                                   Step-size  First-order
 Iter F-count           f(x)                  optimality
    0       3        55.0041            1
    1       6        21.0161       0.2124          17.2
    2       9        11.5085       0.1272          6.08
    3      12        9.56468      0.06553          1.99
    4      15        9.27666      0.02744         0.442
    5      18        9.27666      0.00717         0.442
    6      21        9.27131     0.001793         0.356
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

pOpt =

        Name: 'C1'
       Value: 1.1600e-04
     Minimum: 0
     Maximum: Inf
        Free: 1
       Scale: 0.0020
        Info: [1x1 struct]


1x1 param.Continuous
```

Compare the Measured Output and the Simulated Output

Update the experiment with the estimated capacitance value.

```
Exp = setEstimatedValues(Exp,pOpt);
```

Create a simulation scenario using the experiment and obtain the simulated output.

```
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);
```

Search for the voltage signal in the logged simulation data.

```
SimLog  = find(Simulator.LoggedData,get_param('sdoRCCircuit','SignalLoggingName'));
Voltage = find(SimLog,'Voltage');
```

Plot the measured and simulated data.

The simulated and measured signals match well, except for near time zero. This mismatch is because the capacitor initial voltage defined in the model does not match the initial voltage from the experiment.

```
plot(time,data,'ro',Voltage.Values.Time,Voltage.Values.Data,'b')
title('Simulated and Measured Responses After Estimation')
legend('Measured Voltage','Simulated Voltage')
```
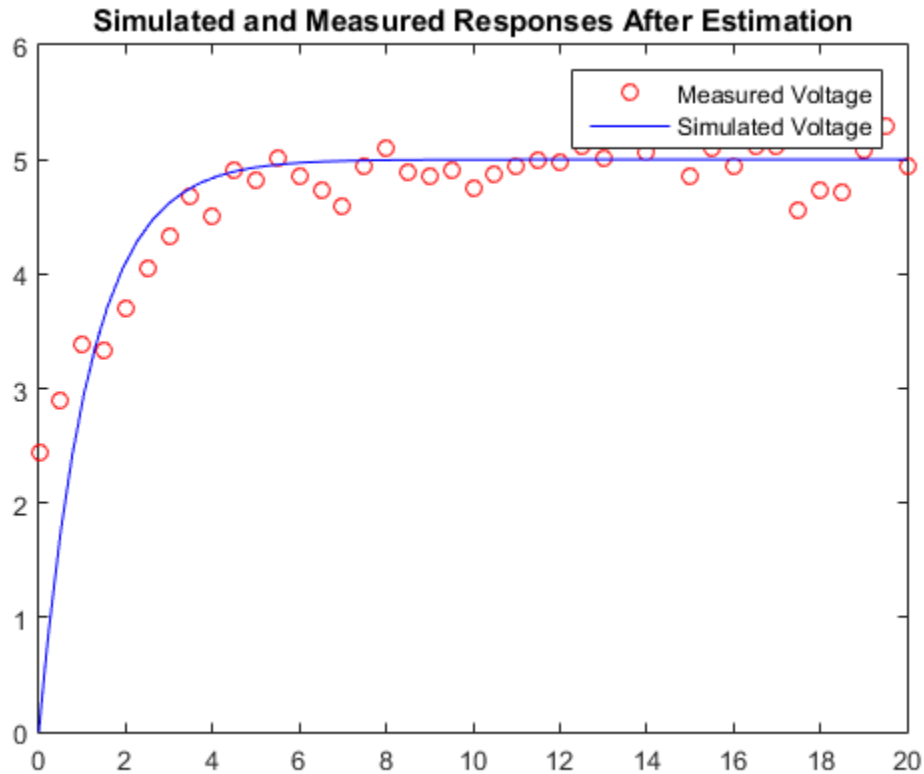
Estimate the Initial State

Add the capacitor initial voltage for the C1 block to the experiment. Set its initial guess value to 1 V.

```
Exp.InitialStates = sdo.getStateFromModel('sdoRCCircuit','C1');
Exp.InitialStates.Value = 1;
```

Recreate the estimation function to use the experiment with initial state estimation

```
estFcn = @(v) sdoRCCircuit_Objective(v,Exp);
```

Get the initial state and capacitance value that is to be estimated from the experiment.

```
v = getValuesToEstimate(Exp);
```

Estimate the parameters.

```
vOpt = sdo.optimize(estFcn,v,opt)
```

```
 Optimization started 20-Jan-2015 08:41:56

                              Step-size  First-order
 Iter F-count         f(x)                optimality
    0      5       4.66337           1
    1     10       2.01883       1.533          21
    2     15       1.34889      0.1257      0.0803
    3     20       1.34365      0.0525        0.12
    4     25       1.34363    0.001294    0.000711
Local minimum found.

Optimization completed because the size of the gradient is less than
the selected value of the function tolerance.

vOpt(1,1) =

       Name: 'sdoRCCircuit/C1:sdoRCCircuit.C1.vc'
      Value: 2.3596
    Minimum: -Inf
    Maximum: Inf
       Free: 1
      Scale: 1
    dxValue: 0
     dxFree: 1
       Info: [1x1 struct]


vOpt(2,1) =

       Name: 'C1'
      Value: 2.2638e-04
    Minimum: 0
    Maximum: Inf
       Free: 1
      Scale: 0.0020
       Info: [1x1 struct]
```

```
2x1 param.Continuous
```

Compare the Measured Output and the Final Simulated Output

Update the experiment with the estimated capacitance and capacitor initial voltage values.

```
Exp = setEstimatedValues(Exp,vOpt);
```

Simulate the model with the estimated initial-state and parameter values and compare the simulated output with the experiment data.

```
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);
SimLog    = find(Simulator.LoggedData,get_param('sdoRCCircuit','SignalLoggingName'));
Voltage   = find(SimLog,'Voltage');

plot(time,data,'ro',Voltage.Values.Time,Voltage.Values.Data,'b')
title('Simulated and Measured Responses After Initial State and Model Parameter Estimat
legend('Measured Voltage','Simulated Voltage')
```

Simulated and Measured Responses After Initial State and Model Parameter Estimation
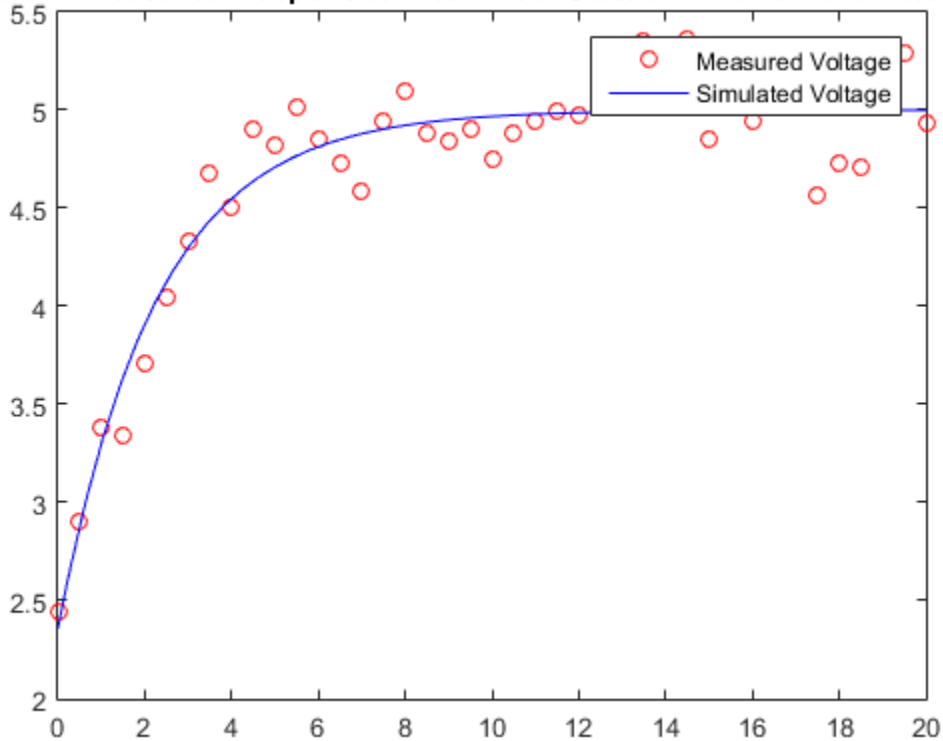
Update the Model Parameter Values

Update the model with the estimated capacitance value. Do not update the model capacitor initial voltage (first element of vOpt) as it is dependent on the experiment.

```
sdo.setValueInModel('sdoRCCircuit',vOpt(2));
```

Related Examples

To learn how to estimate model parameters using the sdo.optimize command, see "Estimate Model Parameters and Initial States (GUI)".

Close the model

```
bdclose('sdoRCCircuit')
```

Estimate Model Parameters using Multiple Experiments (Code)

This example shows how to estimate model parameters from multiple sets of experimental data. You estimate the parameters of a mass-spring-damper system.

Open the Model and Get Experimental Data

This example uses the `sdoMassSpringDamper` model. The model includes two integrators to model the velocity and position of a mass in a mass-spring-damper system.

```
open_system('sdoMassSpringDamper');
```
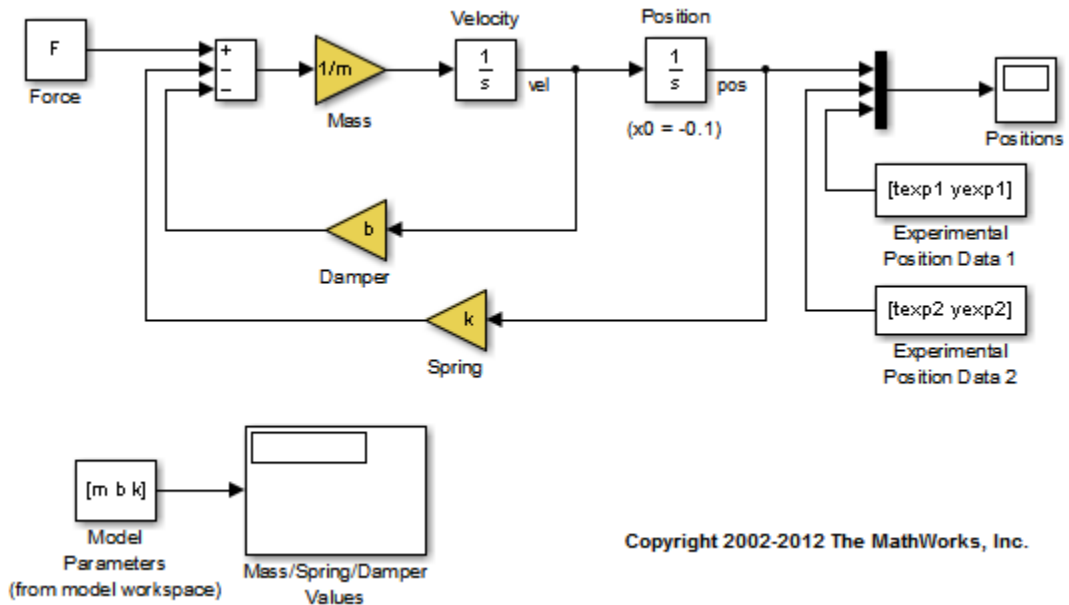

Load the experiment data.

```
load sdoMassSpringDamper_ExperimentData
```

The variables `texp1`, `yexp1`, `texp2`, and `yexp2` are loaded into the workspace. `yexp1` and `yexp2` describe the mass position for times `texp1` and `texp2` respectively.

2-115

Define the Estimation Experiments

Create a 2-element array of experiment objects to store the measured data for the two experiments.

Create an experiment object for the first experiment.

```
Exp = sdo.Experiment('sdoMassSpringDamper');
```

Create an object to store the measured mass position output.

```
MeasuredPos            = Simulink.SimulationData.Signal;
MeasuredPos.Values     = timeseries(yexp1,texp1);
MeasuredPos.BlockPath  = 'sdoMassSpringDamper/Position';
MeasuredPos.PortType   = 'outport';
MeasuredPos.PortIndex  = 1;
MeasuredPos.Name       = 'Position';
```

Add the measured mass position data to the experiment as the expected output data.

```
Exp.OutputData = MeasuredPos;
```

Create an object to specify the initial state for the `Velocity` block. The initial velocity of the mass is 0 m/s.

```
sVel       = sdo.getStateFromModel('sdoMassSpringDamper','Velocity');
sVel.Value = 0;
sVel.Free  = false;
```

`sVel.Free` is set to `false` because the initial velocity is known and does not need to be estimated.

Create an object to specify the initial state for the `Position` block. Specify a guess for the initial mass position. Set the `Free` field of the initial position object to `true` so that it is estimated.

```
sPos       = sdo.getStateFromModel('sdoMassSpringDamper','Position');
sPos.Free  = true;
sPos.Value = -0.1;
```

Add the initial states to the experiment.

```
Exp.InitialStates = [sVel;sPos];
```

Create a 2-element array of experiments. As the two experiments are identical except for the expected output data, copy the first experiment twice.

```
Exp = [Exp; Exp];
```

Modify the expected output data of the second experiment object in Exp.

```
Exp(2).OutputData.Values  = timeseries(yexp2,texp2);
```

Compare the Measured Output and the Initial Simulated Output

Create a simulation scenario using the first experiment and obtain the simulated output.

```
Simulator = createSimulator(Exp(1));
Simulator = sim(Simulator);
```

Search for the position signal in the logged simulation data.

```
SimLog   = find(Simulator.LoggedData,get_param('sdoMassSpringDamper','SignalLoggingName
Position = find(SimLog,'Position');
```

Obtain the simulated position signal for the second experiment.

```
Simulator   = createSimulator(Exp(2));
Simulator   = sim(Simulator);
SimLog      = find(Simulator.LoggedData,get_param('sdoMassSpringDamper','SignalLogging
Position(2) = find(SimLog,'Position');
```

Plot the measured and simulated data.

The model response does not match the experimental output data.

```
subplot(211)
plot(...
    Position(1).Values.Time,Position(1).Values.Data, ...
    Exp(1).OutputData.Values.Time, Exp(1).OutputData.Values.Data,'--')
title('Experiment 1: Simulated and Measured Responses Before Estimation')
ylabel('Position')
legend('Measured Position','Simulated Position','Location','SouthEast')
subplot(212)
plot(...
    Position(2).Values.Time,Position(2).Values.Data, ...
    Exp(2).OutputData.Values.Time, Exp(2).OutputData.Values.Data,'--')
title('Experiment 2: Simulated and Measured Responses Before Estimation')
xlabel('Time (seconds)')
ylabel('Position')
```

```
legend('Measured Position','Simulated Position','Location','SouthEast')
```

Experiment 1: Simulated and Measured Responses Before Estimation

Experiment 2: Simulated and Measured Responses Before Estimation
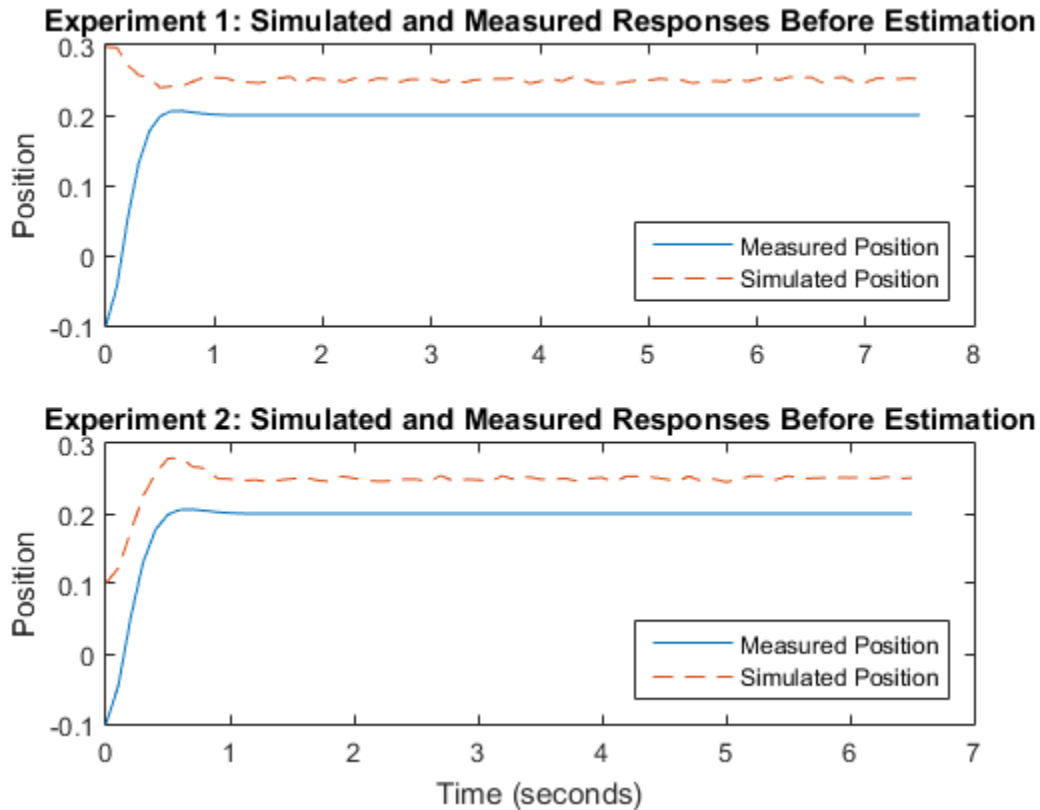
Specify Parameters to Estimate

Select the mass m, spring constant k, and damping coefficient b parameters from the model. Specify that the estimated values for these parameters must be positive.

```
p = sdo.getParameterFromModel('sdoMassSpringDamper', {'b', 'k', 'm'});
p(1).Minimum = 0;
p(2).Minimum = 0;
p(3).Minimum = 0;
```

Get the position initial state values to be estimated from the experiment.

```
s = getValuesToEstimate(Exp);
```

`s` contains two initial state objects, both for the `Position` block. Each object corresponds to an experiment in `Exp`.

Group the model parameters and initial states to be estimated together.

```
v = [p;s]


v(1,1) =

        Name: 'b'
       Value: 100
     Minimum: 0
     Maximum: Inf
        Free: 1
       Scale: 128
        Info: [1x1 struct]


v(2,1) =

        Name: 'k'
       Value: 500
     Minimum: 0
     Maximum: Inf
        Free: 1
       Scale: 512
        Info: [1x1 struct]


v(3,1) =

        Name: 'm'
       Value: 8
     Minimum: 0
     Maximum: Inf
        Free: 1
       Scale: 8
        Info: [1x1 struct]


v(4,1) =

        Name: 'sdoMassSpringDamper/Position'
       Value: -0.1000
```

```
     Minimum: -Inf
     Maximum: Inf
        Free: 1
       Scale: 0.1250
      dxValue: 0
       dxFree: 1
        Info: [1x1 struct]


v(5,1) =

        Name: 'sdoMassSpringDamper/Position'
       Value: -0.1000
     Minimum: -Inf
     Maximum: Inf
        Free: 1
       Scale: 0.1250
      dxValue: 0
       dxFree: 1
        Info: [1x1 struct]


5x1 param.Continuous
```

Define the Estimation Objective

Create an estimation objective function to evaluate how closely the simulation output, generated using the estimated parameter values, matches the measured data.

Use an anonymous function with one input argument that calls the sdoMassSpringDamper_Objective function. We pass the anonymous function to sdo.optimize, which evaluates the function at each optimization iteration.

```
estFcn = @(v) sdoMassSpringDamper_Objective(v,Exp);
```

The sdoMassSpringDamper_Objective function:

- Has one input argument that specifies the mass, spring constant and damper values as well as the initial mass position.

- Has one input argument that specifies the experiment object containing the measured data.

- Returns a vector of errors between simulated and experimental outputs.

The `sdoMassSpringDamper_Objective` function requires two inputs, but `sdo.optimize` requires a function with one input argument. To work around this, `estFcn` is an anonymous function with one input argument, `v`, but it calls `sdoMassSpringDamper_Objective` using two input arguments, `v` and `Exp`.

For more information regarding anonymous functions, see "Anonymous Functions".

The `sdo.optimize` command minimizes the return argument of the anonymous function `estFcn`, that is, the residual errors returned by `sdoMassSpringDamper_Objective`. For more details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

To examine the estimation objective function in more detail, type `edit sdoMassSpringDamper_Objective` at the MATLAB command prompt.

type sdoMassSpringDamper_Objective

```
function vals = sdoMassSpringDamper_Objective(v,Exp)
%SDOMASSSPRINGDAMPER_OBJECTIVE
%
%    The sdoMassSpringDamper_Objective function is used to compare model
%    outputs against experimental data.
%
%    vals = sdoMassSpringDamper_Objective(v,Exp)
%
%    The |v| input argument is a vector of estimated model parameter values
%    and initial states.
%
%    The |Exp| input argument contains the estimation experiment data.
%
%    The |vals| return argument contains information about how well the
%    model simulation results match the experimental data and is used by
%    the |sdo.optimize| function to estimate the model parameters.
%
%    see also sdo.optimize, sdoExampleCostFunction
%

% Copyright 2012 The MathWorks, Inc.

%%
% Define a signal tracking requirement to compute how well the model output
% matches the experiment data. Configure the tracking requirement so that
```

```
% it returns the tracking error residuals (rather than the
% sum-squared-error) and does not normalize the errors.
%
r = sdo.requirements.SignalTracking;
r.Type      = '==';
r.Method    = 'Residuals';
r.Normalize = 'off';

%%
% Update the experiments with the estimated parameter values.
%
Exp  = setEstimatedValues(Exp,v);

%%
% Simulate the model and compare model outputs with measured experiment
% data.
%
Error = [];
for ct=1:numel(Exp)

    Simulator = createSimulator(Exp(ct));
    Simulator = sim(Simulator);

    SimLog  = find(Simulator.LoggedData,get_param('sdoMassSpringDamper','SignalLogging
    Position = find(SimLog,'Position');

    PositionError = evalRequirement(r,Position.Values,Exp(ct).OutputData.Values);

    Error = [Error; PositionError(:)];
end

%%
% Return the residual errors to the optimization solver.
%
vals.F = Error(:);
end
```

Estimate the Parameters

Use the `sdo.optimize` function to estimate the actuator parameter values and initial state.

Specify the optimization options. The estimation function `sdoMassSpringDamper_Objective` returns the error residuals between simulated and

experimental data and does not include any constraints, making this problem ideal for the 'lsqnonlin' solver.

```
opt = sdo.OptimizeOptions;
opt.Method = 'lsqnonlin';
```

Estimate the parameters. Notice that the initial mass position is estimated twice, once for each experiment.

```
vOpt = sdo.optimize(estFcn,v,opt)
```

```
 Optimization started 20-Jan-2015 08:42:18

                                Step-size  First-order
 Iter F-count         f(x)                 optimality
    0     11     0.777696             1
    1     22    0.00413099         3.696      0.00648
    2     33    0.00118327        0.3194      0.00243
    3     44     0.0011106       0.06718     5.09e-05
Local minimum found.

Optimization completed because the size of the gradient is less than
the selected value of the function tolerance.

vOpt(1,1) =

      Name: 'b'
     Value: 58.1959
   Minimum: 0
   Maximum: Inf
      Free: 1
     Scale: 128
      Info: [1x1 struct]


vOpt(2,1) =

      Name: 'k'
     Value: 399.9452
   Minimum: 0
   Maximum: Inf
      Free: 1
     Scale: 512
      Info: [1x1 struct]
```

```
vOpt(3,1) =

        Name: 'm'
       Value: 9.7225
     Minimum: O
     Maximum: Inf
        Free: 1
       Scale: 8
        Info: [1x1 struct]


vOpt(4,1) =

        Name: 'sdoMassSpringDamper/Position'
       Value: 0.2995
     Minimum: -Inf
     Maximum: Inf
        Free: 1
       Scale: 0.1250
     dxValue: O
      dxFree: 1
        Info: [1x1 struct]


vOpt(5,1) =

        Name: 'sdoMassSpringDamper/Position'
       Value: 0.0994
     Minimum: -Inf
     Maximum: Inf
        Free: 1
       Scale: 0.1250
     dxValue: O
      dxFree: 1
        Info: [1x1 struct]


5x1 param.Continuous
```

Compare the Measured Output and the Final Simulated Output

Update the experiments with the estimated parameter values.

```
Exp  = setEstimatedValues(Exp,vOpt);
```

Obtain the simulated output for the first experiment.

```
Simulator   = createSimulator(Exp(1));
Simulator   = sim(Simulator);
SimLog      = find(Simulator.LoggedData,get_param('sdoMassSpringDamper','SignalLoggingN
Position(1) = find(SimLog,'Position');
```

Obtain the simulated output for the second experiment.

```
Simulator   = createSimulator(Exp(2));
Simulator   = sim(Simulator);
SimLog      = find(Simulator.LoggedData,get_param('sdoMassSpringDamper','SignalLoggingN
Position(2) = find(SimLog,'Position');
```

Plot the measured and simulated data.

The model response using the estimated parameter values nicely matches the output
data for the experiments.

```
subplot(211)
plot(...
    Position(1).Values.Time,Position(1).Values.Data, ...
    Exp(1).OutputData.Values.Time, Exp(1).OutputData.Values.Data,'--')
title('Experiment 1: Simulated and Measured Responses After Estimation')
ylabel('Position')
legend('Measured Position','Simulated Position','Location','NorthEast')
subplot(212)
plot(...
    Position(2).Values.Time,Position(2).Values.Data, ...
    Exp(2).OutputData.Values.Time, Exp(2).OutputData.Values.Data,'--')
title('Experiment 2: Simulated and Measured Responses After Estimation')
xlabel('Time (seconds)')
ylabel('Voltage')
legend('Measured Position','Simulated Position','Location','SouthEast')
```

Experiment 1: Simulated and Measured Responses After Estimation

Experiment 2: Simulated and Measured Responses After Estimation
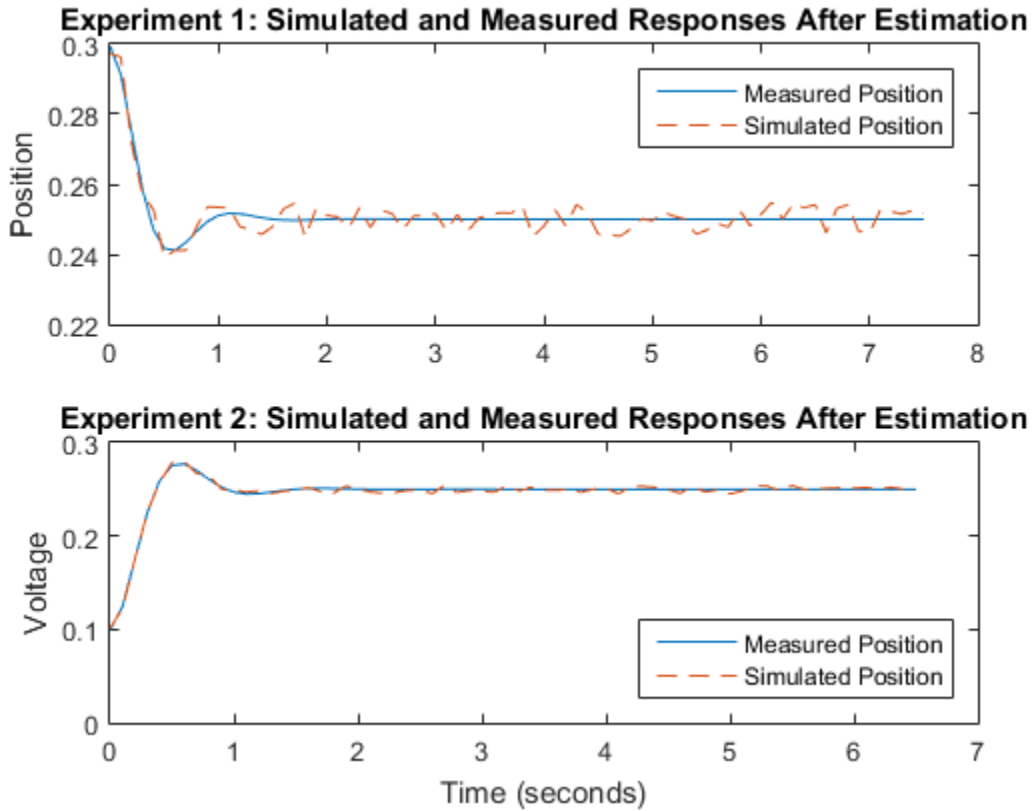
Update the Model Parameter Values

Update the model m, k, and b parameter values. Do not update the model initial position value as this is dependent on the experiment.

```
sdo.setValueInModel('sdoMassSpringDamper',vOpt(1:3));
```

Close the model

```
bdclose('sdoMassSpringDamper')
```

Estimate Model Parameters Per Experiment (Code)

This example shows how to use multiple experiments to estimate a mix of model parameter values; some that are estimated using all the experiments and others that are estimated using individual experiments. The example also shows how to configure estimation experiments with experiment dependent parameter values.

You estimate the parameters of a rechargeable battery based on data collected in experiments that discharge and charge the battery.

Open the Model and Get Experimental Data

This example estimates parameters of a simple, rechargeable battery model, sdoBattery. The model input is the battery current and the model output, the battery terminal voltage, is computed from the battery state-of-charge.

```
open_system('sdoBattery');
```

Simple Battery Model

Copyright 2012-2014 The MathWorks, Inc.

The model is based on the equation

$$E = (1 - Loss) * V - K * Q_{max} * \frac{1 - s}{s}$$

Where:

- E is the battery terminal voltage in Volts.

- V is the battery constant voltage in Volts.

- K is the battery polarization resistance in Ohms.

- Q_{max} is the maximum battery capacity in Ampere-Hour.

- s is the battery charge state, with 1 being fully charged and 0 zero charge. The battery state-of-charge is computed from the integral of the battery current with a +ve current indicating discharge and a -ve current indicating charging. The battery initial state-of-charge is specified by Q_0 in Ampere-Hour.

- $Loss$ is the voltage drop when charging, expressed as a fraction of the battery constant voltage. When the battery is discharging this value is zero.

V, K, Qmax, Q0, and Loss are variables defined in the model workspace.

Load the experiment data. A 1.2V (6500mAh) battery was subjected to a discharge experiment and a charging experiment.

```
load sdoBattery_ExperimentData
```

The variables Charge_Data and DCharge_Data are loaded into the workspace. The first column of Charge_Data contains time data. The second and third columns of Charge_Data describe the current and voltage during a battery charging experiment. DCharge_Data is similarly structured and contains data for a battery discharging experiment.

Plot the Experiment Data

```
subplot(221),
plot(DCharge_Data(:,1)/3600,DCharge_Data(:,2))
title('Experiment: Discharge')
xlabel('Time (hours)')
ylabel('Current (A)')
subplot(223)
plot(DCharge_Data(:,1)/3600,DCharge_Data(:,3))
xlabel('Time (hours)')
ylabel('Voltage (V)')
subplot(222),
plot(Charge_Data(:,1)/3600,Charge_Data(:,2))
title('Experiment: Charge')
xlabel('Time (hours)')
```

```matlab
ylabel('Current (A)')
subplot(224)
plot(Charge_Data(:,1)/3600,Charge_Data(:,3))
xlabel('Time (hours)')
ylabel('Voltage (V)')
```
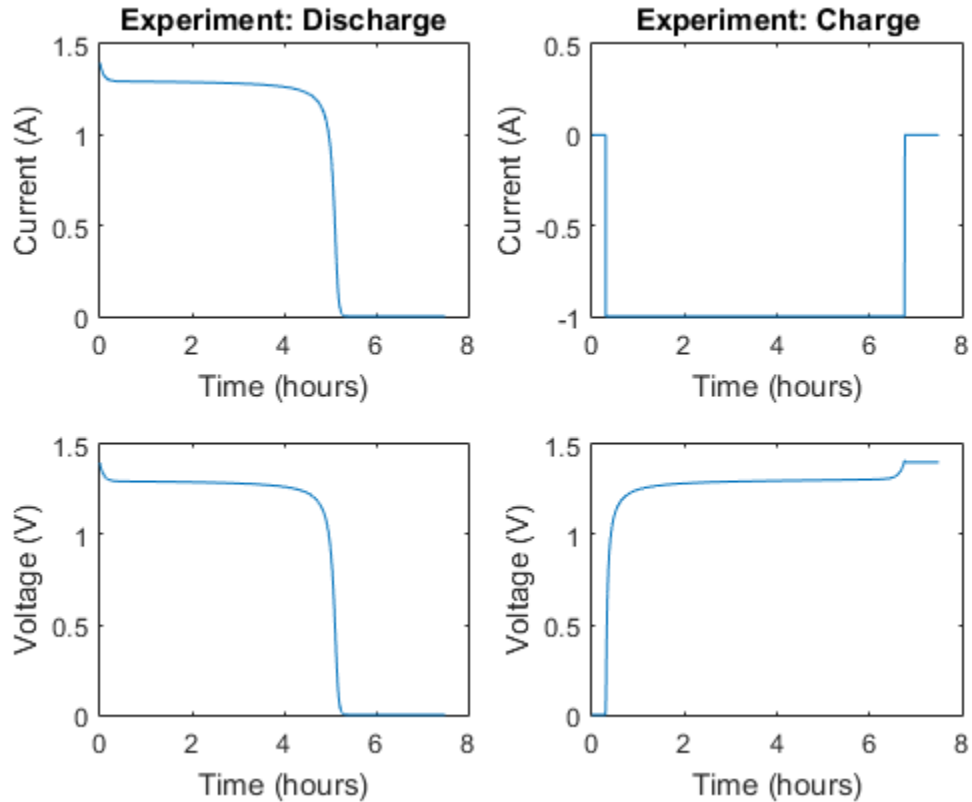

Define the Estimation Experiments

Create a 2-element array of experiment objects to specify the measured data for the two experiments.

Create an experiment object for the battery discharge experiment. The measured current data is specified as a timeseries in the experiment object.

```matlab
DCharge_Exp = sdo.Experiment('sdoBattery');
```

Specify the input data (current) as a timeseries object.

```
DCharge_Exp.InputData = timeseries(DCharge_Data(:,2),DCharge_Data(:,1));
```

Create an object to specify the measured voltage output data.

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

Add the voltage signal to the discharge experiment as the expected output data.

```
DCharge_Exp.OutputData = VoltageSig;
```

Specify the battery initial charge state for the experiment. The battery charge state is modeled by the Q (Ah) block and it's initial value is specified by the variable Q0. Create a parameter for the Q0 variable and add the parameter to the experiment. Q0 is experiment dependent and assumes different values in the discharging and charging experiments.

```
Q0 = sdo.getParameterFromModel('sdoBattery','Q0');
Q0.Value = 6.5;
Q0.Free  = false;
```

Q0.Free is set to false because the initial battery charge is known and does not need to be estimated.

Add the Q0 parameter to the experiment.

```
DCharge_Exp.Parameters = Q0;
```

Create an experiment object to store the charging experiment data. Add the measured current input and measured voltage output data to the object.

```
Charge_Exp = sdo.Experiment('sdoBattery');
Charge_Exp.InputData = timeseries(Charge_Data(:,2),Charge_Data(:,1));
VoltageSig.Values    = timeseries(Charge_Data(:,3),Charge_Data(:,1));
Charge_Exp.OutputData = VoltageSig;
```

Add the battery initial charge and charging loss fraction parameters to the experiment. For this experiment, the initial charge (Q0) is known (0 Ah), but the value of the charging loss fraction (Loss) is not known.

```
Q0.Value = 0;

Loss = sdo.getParameterFromModel('sdoBattery','Loss');
Loss.Free    = true;
Loss.Minimum = 0;
Loss.Maximum = 0.5;

Charge_Exp.Parameters = [Q0;Loss];
```

`Loss.Free` is set to true so that the value of `Loss` is estimated.

Collect both experiments into one vector.

```
Exp = [DCharge_Exp; Charge_Exp];
```

Compare the Measured Output and the Initial Simulated Output

Create a simulation scenario using the first (discharging) experiment and obtain the simulated output.

```
Simulator  = createSimulator(Exp(1));
Simulator  = sim(Simulator);
```

Search for the voltage signal in the logged simulation data.

```
SimLog     = find(Simulator.LoggedData,get_param('sdoBattery','SignalLoggingName'));
Voltage(1) = find(SimLog,'Voltage');
```

Obtain the simulated voltage signal for the second (charging) experiment.

```
Simulator  = createSimulator(Exp(2));
Simulator  = sim(Simulator);
SimLog     = find(Simulator.LoggedData,get_param('sdoBattery','SignalLoggingName'));
Voltage(2) = find(SimLog,'Voltage');
```

Plot the measured and simulated data.

The model response does not match the experimental output data.

```
subplot(211)
plot(...
    Voltage(1).Values.Time/3600,Voltage(1).Values.Data, ...
    Exp(1).OutputData.Values.Time/3600, Exp(1).OutputData.Values.Data,'-.')
title('Discharging Experiment: Simulated and Measured Responses Before Estimation')
ylabel('Voltage (V)')
legend('Simulated Voltage','Measured Voltage','Location','SouthWest')
```

2-131

```
subplot(212)
plot(...
    Voltage(2).Values.Time/3600,Voltage(2).Values.Data, ...
    Exp(2).OutputData.Values.Time/3600, Exp(2).OutputData.Values.Data,'-.')
title('Charging Experiment: Simulated and Measured Responses Before Estimation')
xlabel('Time (hours)')
ylabel('Voltage (V)')
legend('Simulated Voltage','Measured Voltage','Location','SouthEast')
```


Specify Parameters to Estimate

Estimate the values of the battery voltage V, the battery polarization resistance K, and the charging loss fraction Loss. The V and K parameters are estimated using all the experiment data while the Loss parameter is estimated using only the charging data.

Select the battery voltage V and the battery polarization resistance K parameters from the model. Specify minimum and maximum bounds for these parameters.

```
p = sdo.getParameterFromModel('sdoBattery',{'V','K'});

p(1).Minimum = 0;
p(1).Maximum = 2;

p(2).Minimum = 1e-6;
p(2).Maximum = 1e-1;
```

Get the experiment-specific Loss parameter from the experiment.

```
s = getValuesToEstimate(Exp);
```

Group all the parameters to be estimated.

```
v = [p;s]


v(1,1) =

      Name: 'V'
     Value: 1.2000
   Minimum: 0
   Maximum: 2
      Free: 1
     Scale: 2
      Info: [1x1 struct]


v(2,1) =

      Name: 'K'
     Value: 1.0000e-03
   Minimum: 1.0000e-06
   Maximum: 0.1000
      Free: 1
     Scale: 0.0020
      Info: [1x1 struct]


v(3,1) =

      Name: 'Loss'
     Value: 0.0100
```

```
    Minimum: 0
    Maximum: 0.5000
       Free: 1
      Scale: 0.0156
       Info: [1x1 struct]


3x1 param.Continuous
```

Define the Estimation Objective

Create an estimation objective function to evaluate how closely the simulation output, generated using the estimated parameter values, matches the measured data.

Use an anonymous function with one input argument that calls the `sdoBattery_Objective` function. We pass the anonymous function to `sdo.optimize`, which evaluates the function at each optimization iteration.

```
estFcn = @(v) sdoBattery_Objective(v,Exp);
```

The `sdoBattery_Objective` function:

- Has one input argument that specifies the estimated battery parameter values.

- Has one input argument that specifies the experiment object containing the measured data.

- Returns a vector of errors between simulated and experimental outputs.

The `sdoBattery_Objective` function requires two inputs, but `sdo.optimize` requires a function with one input argument. To work around this, `estFcn` is an anonymous function with one input argument, `v`, but it calls `sdoBattery_Objective` using two input arguments, `v` and `Exp`.

For more information regarding anonymous functions, see "Anonymous Functions".

The `sdo.optimize` command minimizes the return argument of the anonymous function `estFcn`, that is, the residual errors returned by `sdoBattery_Objective`. For more details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

To examine the estimation objective function in more detail, type `edit sdoBattery_Objective` at the MATLAB command prompt.

```
type sdoBattery_Objective

function vals = sdoBattery_Objective(v,Exp)
%SDOBATTERY_OBJECTIVE
%
%    The sdoBattery_Objective function is used to compare model
%    outputs against experimental data.
%
%    vals = sdoBattery_Objective(v,Exp)
%
%    The |v| input argument is a vector of estimated model parameter values
%    and initial states.
%
%    The |Exp| input argument contains the estimation experiment data.
%
%    The |vals| return argument contains information about how well the
%    model simulation results match the experimental data and is used by
%    the |sdo.optimize| function to estimate the model parameters.
%
%    See also sdo.optimize, sdoExampleCostFunction, sdoBattery_cmddemo
%

% Copyright 2012 The MathWorks, Inc.

%%
% Define a signal tracking requirement to compute how well the model output
% matches the experiment data. Configure the tracking requirement so that
% it returns the tracking error residuals (rather than the
% sum-squared-error) and does not normalize the errors.
%
r = sdo.requirements.SignalTracking;
r.Type      = '==';
r.Method    = 'Residuals';
r.Normalize = 'off';

%%
% Update the experiments with the estimated parameter values.
%
Exp  = setEstimatedValues(Exp,v);

%%
% Simulate the model and compare model outputs with measured experiment
% data.
%
```

```
Error = [];
for ct=1:numel(Exp)

    Simulator = createSimulator(Exp(ct));
    Simulator = sim(Simulator);

    SimLog  = find(Simulator.LoggedData,get_param('sdoBattery','SignalLoggingName'));
    Voltage = find(SimLog,'Voltage');

    VoltageError = evalRequirement(r,Voltage.Values,Exp(ct).OutputData(1).Values);

    Error = [Error; VoltageError(:)];
end

%%
% Return the residual errors to the optimization solver.
%
vals.F = Error(:);
end
```

Estimate the Parameters

Use the `sdo.optimize` function to estimate the battery parameter values.

Specify the optimization options. The estimation function `sdoBattery_Objective` returns the error residuals between simulated and experimental data and does not include any constraints, making this problem ideal for the 'lsqnonlin' solver.

```
opt = sdo.OptimizeOptions;
opt.Method = 'lsqnonlin';
```

Estimate the parameters.

```
vOpt = sdo.optimize(estFcn,v,opt)


 Optimization started 20-Jan-2015 08:42:48

                             Step-size  First-order
 Iter F-count        f(x)                optimality
    0      7       3272.22          1
    1     14       619.356     0.1634     3.15e+05
    2     21       411.131     0.2175         28.7
    3     28       405.529     0.3838     2.16e+03
    4     35       403.727     0.2767         15.2
    5     42       403.379     0.1645     1.14e+03
```

```
Local minimum possible.

lsqnonlin stopped because the final change in the sum of squares relative to
its initial value is less than the selected value of the function tolerance.

vOpt(1,1) =

        Name: 'V'
       Value: 1.3083
     Minimum: 0
     Maximum: 2
        Free: 1
       Scale: 2
        Info: [1x1 struct]


vOpt(2,1) =

        Name: 'K'
       Value: 0.0010
     Minimum: 1.0000e-06
     Maximum: 0.1000
        Free: 1
       Scale: 0.0020
        Info: [1x1 struct]


vOpt(3,1) =

        Name: 'Loss'
       Value: 5.1801e-05
     Minimum: 0
     Maximum: 0.5000
        Free: 1
       Scale: 0.0156
        Info: [1x1 struct]


3x1 param.Continuous
```

Compare the Measured Output and the Final Simulated Output

Update the experiments with the estimated parameter values.

```
Exp  = setEstimatedValues(Exp,vOpt);
```

Obtain the simulated output for the first (discharging) experiment.

```
Simulator  = createSimulator(Exp(1));
Simulator  = sim(Simulator);
SimLog     = find(Simulator.LoggedData,get_param('sdoBattery','SignalLoggingName'));
Voltage(1) = find(SimLog,'Voltage');
```

Obtain the simulated output for the second (charging) experiment.

```
Simulator  = createSimulator(Exp(2));
Simulator  = sim(Simulator);
SimLog     = find(Simulator.LoggedData,get_param('sdoBattery','SignalLoggingName'));
Voltage(2) = find(SimLog,'Voltage');
```

Plot the measured and simulated data.

The simulation results match the experimental data well except in the regions when the battery is fully charged. This is not unexpected as the simple battery model does not model the exponential voltage drop when the battery is fully charged.

```
subplot(211)
plot(...
    Voltage(1).Values.Time/3600,Voltage(1).Values.Data, ...
    Exp(1).OutputData.Values.Time/3600, Exp(1).OutputData.Values.Data,'-.')
title('Discharging Experiment: Simulated and Measured Responses After Estimation')
ylabel('Voltage (V)')
legend('Simulated Voltage','Measured Voltage','Location','SouthWest')
subplot(212)
plot(...
    Voltage(2).Values.Time/3600,Voltage(2).Values.Data, ...
    Exp(2).OutputData.Values.Time/3600, Exp(2).OutputData.Values.Data,'-.')
title('Charging Experiment: Simulated and Measured Responses After Estimation')
xlabel('Time (hours)')
ylabel('Voltage (V)')
legend('Simulated Voltage','Measured Voltage','Location','SouthEast')
```

Discharging Experiment: Simulated and Measured Responses After Estimation
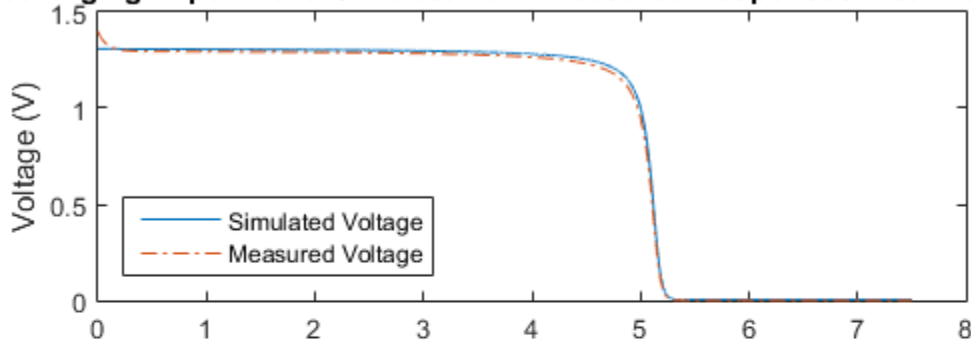
Charging Experiment: Simulated and Measured Responses After Estimation
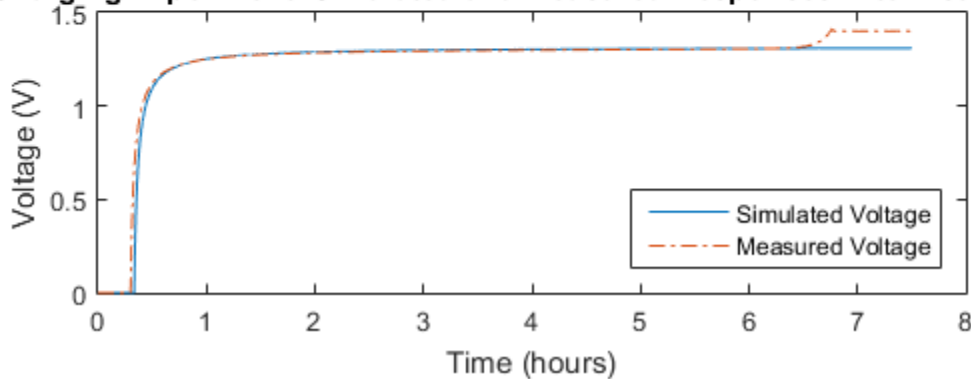
Update the Model Parameter Values

Update the model V, K, and Loss parameter values.

```
sdo.setValueInModel('sdoBattery',vOpt);
```

Related Examples

To learn how to estimate the battery parameters using the Parameter Estimation Tool, see "Estimate Model Parameters Per Experiment (GUI)".

Close the model

```
bdclose('sdoBattery')
```

2-139

Estimate Model Parameters with Parameter Constraints (Code)

This example shows how to estimate model parameters while imposing constraints on the parameter values.

You estimate dynamic and static friction coefficients of a simple friction system.

Open the Model and Get Experimental Data

This example estimates parameters for a simple friction system, `sdoFriction`. The model input is the force applied to a mass and the model outputs are the mass position and velocity.

```
open_system('sdoFriction');
```


The model is based on a mass sliding on a surface. The mass is subject to a static friction that must be overcome before the mass moves and a dynamic friction once the mass moves. The static friction, `u_static`, is a fraction of the mass normal force; similarly the dynamic friction, `u_dynamic`, is a fraction of the mass normal force.

Load the experiment data. The mass was subjected to an applied force and its position recorded.

```
load sdoFriction_ExperimentData
```

The variables `AppliedForce`, `Position`, and `Velocity` are loaded into the workspace. The first column of each of these variables represents time and the second column represents the measured data. Because velocity is the first derivative of position, we only use the position measurements for this example.

Plot the Experiment Data

```
subplot(211),
plot(AppliedForce(:,1),AppliedForce(:,2))
title('Measured Applied Force Input for Simple Friction System');
ylabel('Applied Force (N)')
subplot(212)
plot(Position(:,1),Position(:,2))
title('Measured Mass Position for Simple Friction System');
xlabel('Time (seconds)')
ylabel('Position (m)')
```
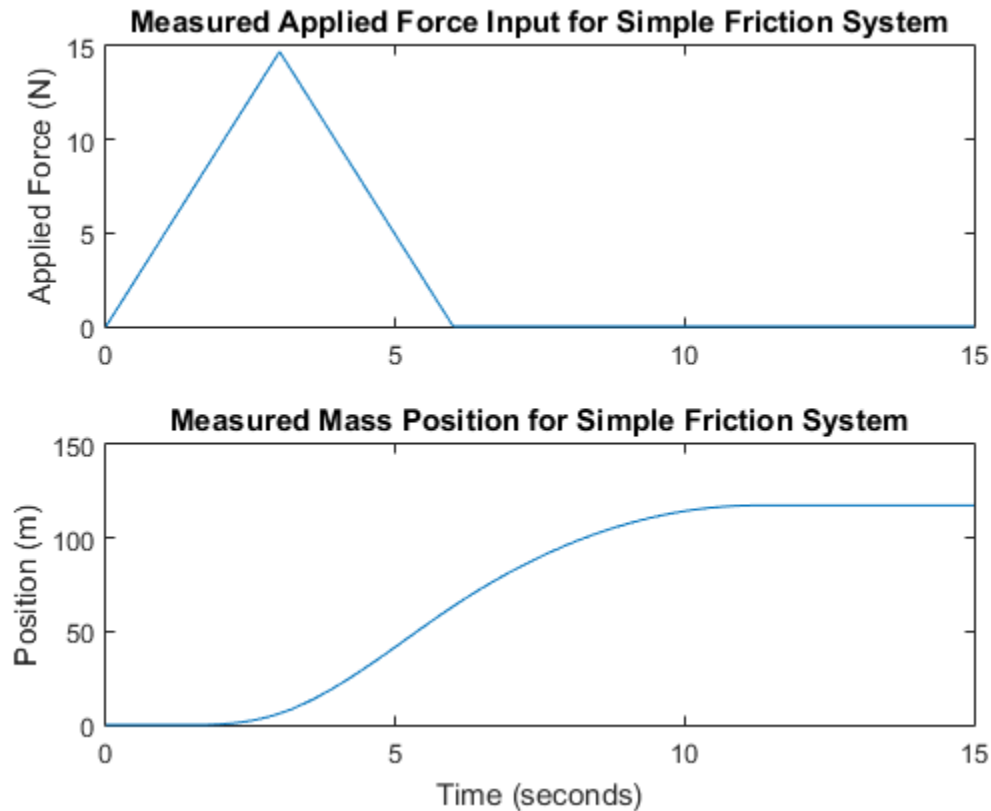
Define the Estimation Experiment

Create an experiment object to specify the experiment data.

```
Exp = sdo.Experiment('sdoFriction');
```

Specify the input data (applied force) as a timeseries object.

```
Exp.InputData = timeseries(AppliedForce(:,2),AppliedForce(:,1));
```

Create an object to specify the measured mass position output.

```
PositionSig = Simulink.SimulationData.Signal;
```

```
PositionSig.Name      = 'Position';
PositionSig.BlockPath = 'sdoFriction/x';
PositionSig.PortType  = 'outport';
PositionSig.PortIndex = 1;
PositionSig.Values    = timeseries(Position(:,2),Position(:,1));
```

Add the measured mass position data to the experiment as the expected output data.

```
Exp.OutputData = PositionSig;
```

Compare the Measured Output and the Initial Simulated Output

Create a simulation scenario using the experiment and obtain the simulated output.

```
Simulator    = createSimulator(Exp);
Simulator    = sim(Simulator);
```

Search for the position signal in the logged simulation data.

```
SimLog    = find(Simulator.LoggedData,get_param('sdoFriction','SignalLoggingName'));
Position  = find(SimLog,'Position');
```

Plot the measured and simulated data.

As expected, the model response does not match the experimental output data.

```
figure
plot(...
    Position.Values.Time,Position.Values.Data, ...
    Exp.OutputData.Values.Time, Exp.OutputData.Values.Data,'-.')
title('Simulated and Measured Responses Before Estimation')
ylabel('Position (m)')
xlabel('Time (seconds)')
legend('Simulated Position','Measured Position','Location','NorthWest')
```

Simulated and Measured Responses Before Estimation
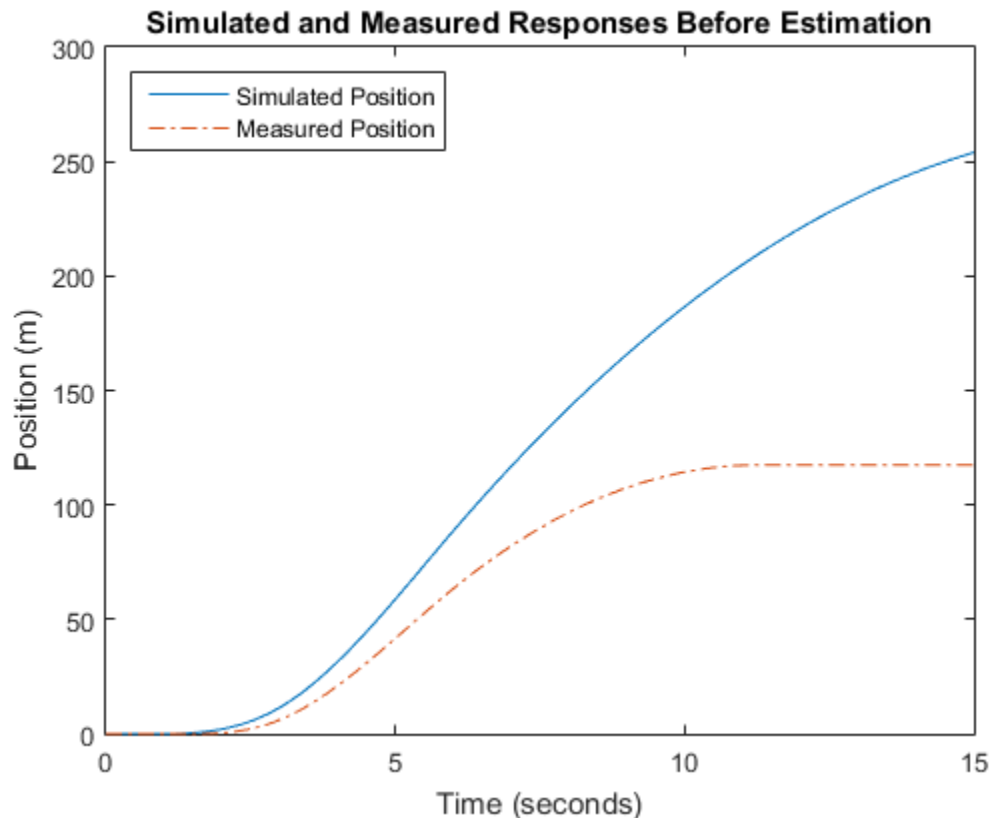
Specify Parameters to Estimate

Estimate the u_static and u_dynamic friction coefficients using the experiment data. These coefficients are used as gains in the Static Friction and Dynamic Friction blocks, respectively. Physics indicates that friction coefficients should be constrained so that u_static $\geq$ u_dynamic; this parameter constraint is implemented in the estimation objective function.

Select the u_static and u_dynamic model parameters. Specify bounds for the estimated parameter values. Both coefficients are limited to the range [0 1].

```
p = sdo.getParameterFromModel('sdoFriction',{'u_static','u_dynamic'});
```

```
p(1).Minimum = 0;
p(1).Maximum = 1;

p(2).Minimum = 0;
p(2).Maximum = 1;
```

Define the Estimation Objective

Create an estimation objective function to evaluate how closely the simulation output, generated using the estimated parameter values, matches the measured data.

Use an anonymous function with one input argument that calls the `sdoFriction_Objective` function. We pass the anonymous function to `sdo.optimize`, which evaluates the function at each optimization iteration.

```
estFcn = @(v) sdoFriction_Objective(v,Exp);
```

The `sdoFriction_Objective` function:

- Has one input argument that specifies the estimated friction coefficients.

- Has one input argument that specifies the experiment object containing the measured data.

- Returns the sum-squared-error errors between simulated and experimental outputs, and returns the parameter constraint.

The `sdoFriction_Objective` function requires two inputs, but `sdo.optimize` requires a function with one input argument. To work around this, `estFcn` is an anonymous function with one input argument, `v`, but it calls `sdoFriction_Objective` using two input arguments, `v` and `Exp`.

For more information regarding anonymous functions, see "Anonymous Functions".

The `sdo.optimize` command minimizes the return argument of the anonymous function `estFcn`, that is, the residual errors returned by `sdoFriction_Objective`. For more details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

To examine the estimation objective function in more detail, type `edit sdoFriction_Objective` at the MATLAB command prompt.

2-145

```
type sdoFriction_Objective

function vals = sdoFriction_Objective(p,Exp)
%SDOFRICTION_OBJECTIVE
%
%    The sdoFriction_Objective function is used to compare model
%    outputs against experimental data and measure how well constraints are
%    satisfied.
%
%    vals = sdoFriction_Objective(p,Exp)
%
%    The |p| input argument is a vector of estimated model parameter values.
%
%    The |Exp| input argument contains the estimation experiment data.
%
%    The |vals| return argument contains information about how well the
%    model simulation results match the experimental data and how well
%    constraints are satisfied. The |vals| argument is used by the
%    |sdo.optimize| function to estimate the model parameters.
%
%    See also sdo.optimize, sdoExampleCostFunction, sdoFriction_cmddemo
%

% Copyright 2012 The MathWorks, Inc.

%%
% Define a signal tracking requirement to compute how well the model output
% matches the experiment data. Configure the tracking requirement so that
% it returns the sum-squared-error.
%
r = sdo.requirements.SignalTracking;
r.Type   = '==';
r.Method = 'SSE';

%%
% Update the experiments with the estimated parameter values.
%
Exp  = setEstimatedValues(Exp,p);

%%
% Simulate the model and compare model outputs with measured experiment
% data.
%
Simulator = createSimulator(Exp);
```

```
Simulator = sim(Simulator);

SimLog  = find(Simulator.LoggedData,get_param('sdoFriction','SignalLoggingName'));
Position = find(SimLog,'Position');

PositionError = evalRequirement(r,Position.Values,Exp.OutputData(1).Values);

%%
% Measure how well the parameters satisfy the friction coefficient constraint,
% |u_static| >= |u_dynamic|. Note that constraints are returned to the
% optimizer in a c <=O format. The friction coefficient constraint is
% rewritten accordingly.
PConstr =  p(2).Value - p(1).Value; % u_dynamic - u_static <= O

%%
% Return the sum-squared-error and constraint violation to the optimization
% solver.
%
vals.F    = PositionError(:);
vals.Cleq = PConstr;
end
```

The friction coefficient constraint, $\text{u\_static} \geq \text{u\_dynamic}$, is implemented in the sdoFriction_Objective function as $\text{u\_dynamic} - \text{u\_static} \leq 0$. This is because the optimizer requires constraint values in a $c \leq 0$ format. For more information, type `help sdo.optimize` at the MATLAB command prompt.

Estimate the Parameters

Use the `sdo.optimize` function to estimate the friction model parameter values.

Specify the optimization options. The estimation function sdoFriction_Objective returns the sum-squared-error between simulated and experimental data and includes a parameter constraint. The default 'fmincon' solver is ideal for this type of problem.

Estimate the parameters.

```
pOpt = sdo.optimize(estFcn,p)
```

```
 Optimization started 20-Jan-2015 08:43:51

                          max       Step-size   First-order
 Iter F-count      f(x)   constraint             optimality
```

```
 0     5     27.7267           0
 1    11     22.5643           0        2.21         72.9
 2    15     17.4771           0        0.51           16
 3    22      0.762174         0        1.33         10.7
 4    29      0.40765          0        0.263        3.15
 5    34      0.0254254        0        0.0897       1.22
 6    39      0.00522001       0        0.0296       0.276
 7    44      0.00398126       0        0.0209       0.185
 8    49      0.00120167       0        0.111        0.17
 9    60      0.00118106       0        0.0212       0.173
10    72      0.00110164       0        0.0262       0.165
11    91      0.00110097       0        0.0031       0.174
12   108      0.00110097       0        0.00165      0.174
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the selected value of the step size tolerance and constraints are
satisfied to within the selected value of the constraint tolerance.

pOpt(1,1) =

        Name: 'u_static'
       Value: 0.7973
     Minimum: 0
     Maximum: 1
        Free: 1
       Scale: 0.5000
        Info: [1x1 struct]


pOpt(2,1) =

        Name: 'u_dynamic'
       Value: 0.4021
     Minimum: 0
     Maximum: 1
        Free: 1
       Scale: 0.2500
        Info: [1x1 struct]


2x1 param.Continuous
```

Compare the Measured Output and the Final Simulated Output

Update the experiments with the estimated parameter values.

```
Exp  = setEstimatedValues(Exp,pOpt);
```

Obtain the simulated output for the experiment.

```
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);
SimLog    = find(Simulator.LoggedData,get_param('sdoFriction','SignalLoggingName'));
Position  = find(SimLog,'Position');
```

Plot the measured and simulated data.

It can be seen that the model response using the estimated parameter values nicely matches the experiment output data.

```
plot(...
    Position.Values.Time,Position.Values.Data, ...
    Exp.OutputData.Values.Time, Exp.OutputData.Values.Data,'-.')
title('Simulated and Measured Responses After Model Parameter Estimation')
ylabel('Position (m)')
xlabel('Time (seconds)')
legend('Simulated Position','Measured Position','Location','NorthWest')
```

Simulated and Measured Responses After Model Parameter Estimation
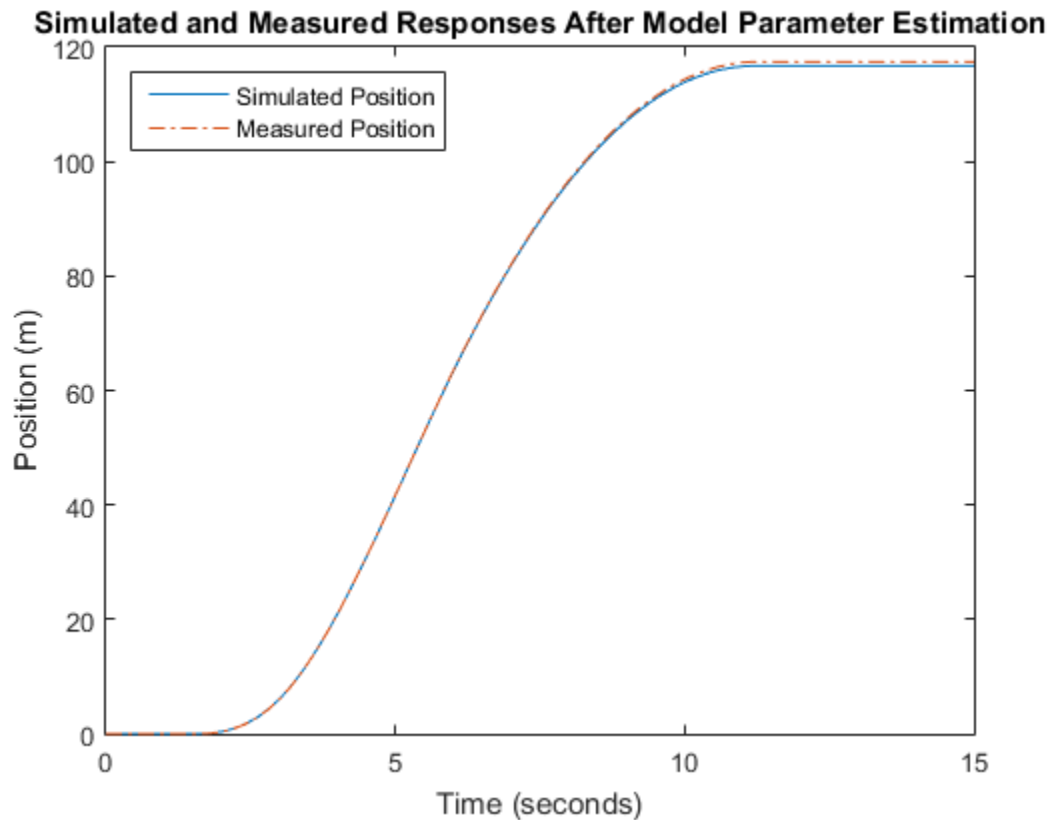
Update the Model Parameter Values

Update the model u_static and u_dynamic parameter values.

```
sdo.setValueInModel('sdoFriction',pOpt);
```

Close the model

```
bdclose('sdoFriction')
```

Estimate Model Parameter Values (GUI)

This example shows how to use experiment data to estimate model parameters. You estimate the parameters of an engine throttle system.

Simulink® Model of the Engine Throttle System

The Simulink® model for the engine throttle system, `spe_engine_throttle`, is shown below.

Engine Throttle Model

Copyright (c) 2002-2014 The MathWorks, Inc.
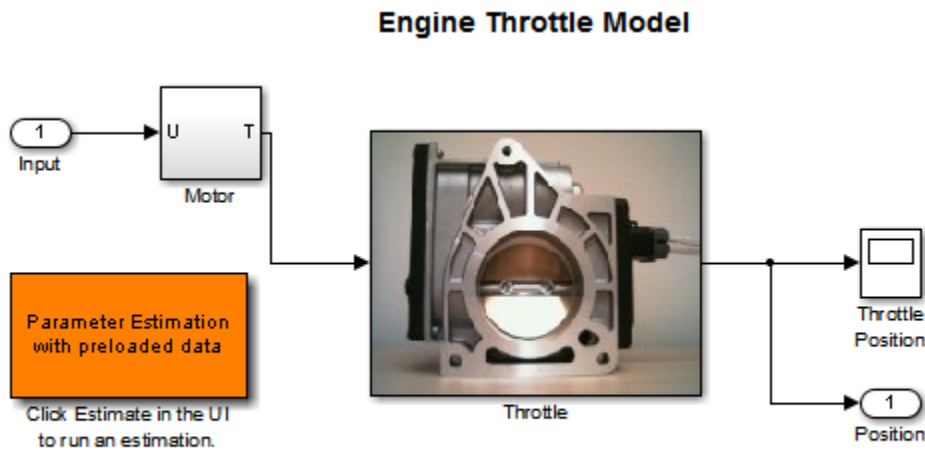
Throttle Model Description

The throttle controls the air mass flow into the intake manifold of an engine. The throttle body contains a butterfly valve that opens when the driver presses down on the accelerator pedal. This lets more air enter the cylinders and causes the engine to produce more torque.

A DC motor controls the opening angle of the butterfly valve. There is also a spring attached to the valve to return it to its closed position when the DC motor is de-

energized. The amount of rotation of the valve is limited to approximately 90 degrees. Therefore, if a large command input is applied to the motor, the valve hits the hard stops preventing it from rotating further.

The motor is modeled as a torque gain and a time-delay input with parameters Kt and input_delay. The butterfly valve is modeled as a mass-spring-damper system with parameters J, c and k. This system is augmented with hard stops to limit the valve opening to 90 degrees. We know the model components, however, the parameter values of the system are not known accurately.

Estimation Experiment Data

Double-click the `Parameter Estimation GUI with preloaded data` block in the model to open a pre-configured estimation GUI session.

The saved estimation project defines three experiments; the `EstimationData` experiment will be used for parameter estimation, while `ValidationData1`, `ValidationData2` are used for validating the estimated parameters. The `EstimateData` experiment is plotted.

The signal data for the experiments can be imported from various sources including MATLAB® variables, MAT files, Excel® files, or comma-separated-value files. See "Importing and Preprocessing Experiment Data (GUI)" for more information.
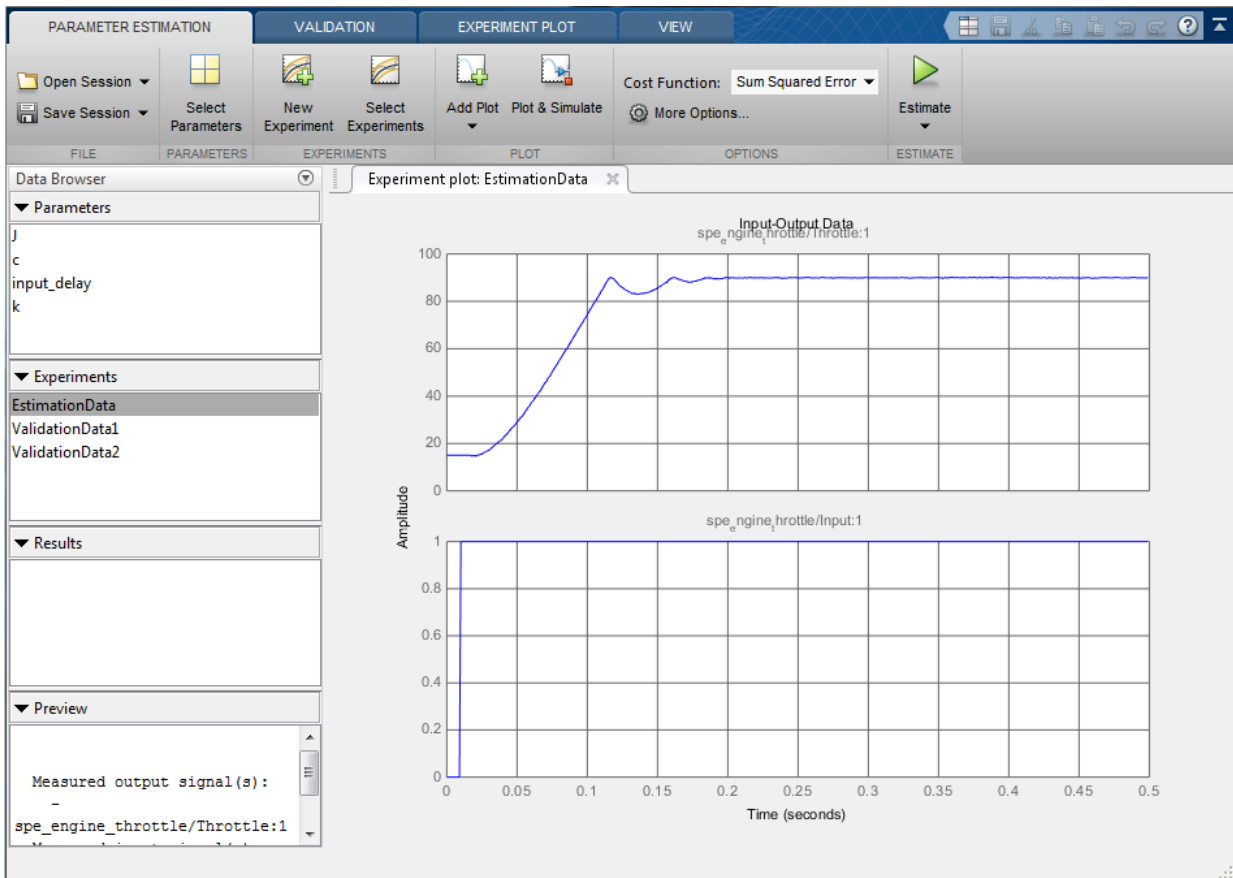
The experiment plot is also used to see how well the measured data matches the current model. Click **Plot Model Response** to display simulated signal data on the experiment plots.
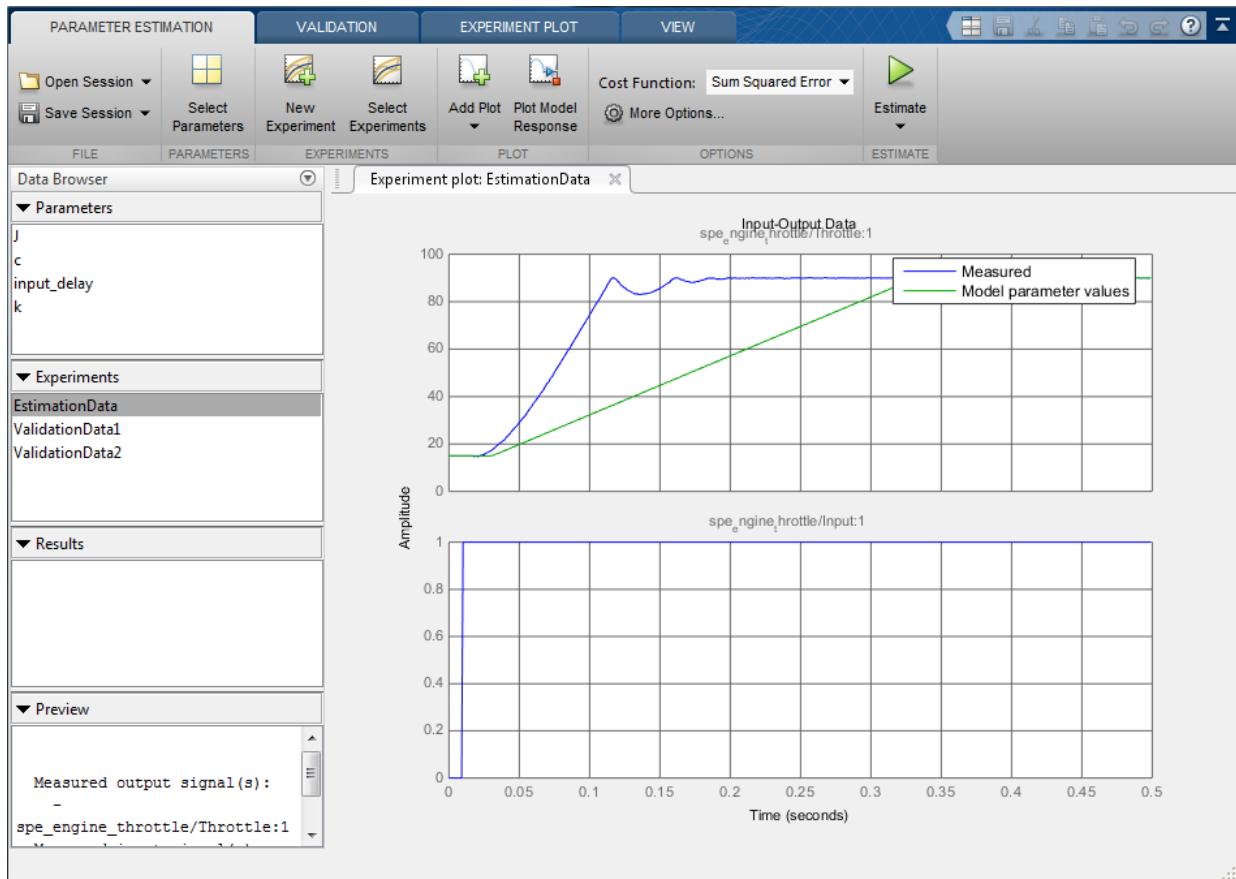
The simulation results show that the model does not match the measured data and that model parameters need to be estimated.

Estimated Parameters

The next step is to define the parameter to estimate. Click **Select Parameters** to open a dialog to select model parameters to estimate. In this example we have preselected the four unknown parameters; the butterfly valve inertia, J; the damping coefficient, c; the return spring constant, k; and the time lag in motor response, input_delay.
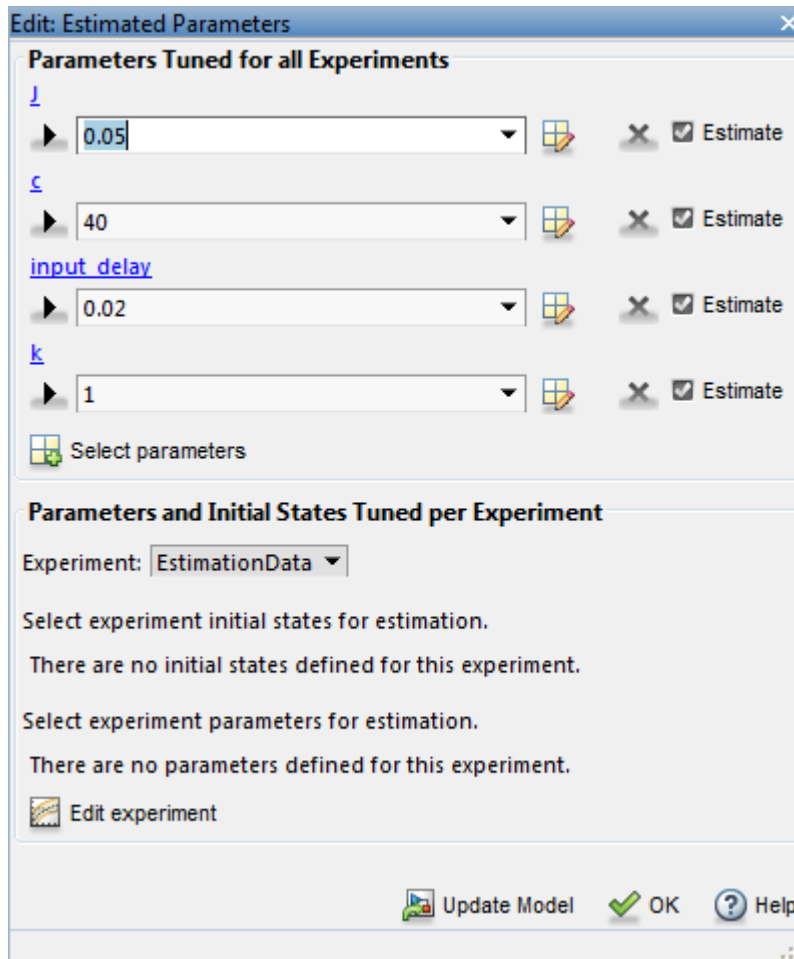
Since we know from physical insight that all of these parameters have positive values, we set their lower limits to zero. We also put an upper bound of 0.1 sec on the input_delay parameter. We can also select an initial value for the parameters. These may come from some quick calculations of some formulas that determine the parameters.

Click the right arrow toggle button to modify the parameter minimum and maximum bounds.
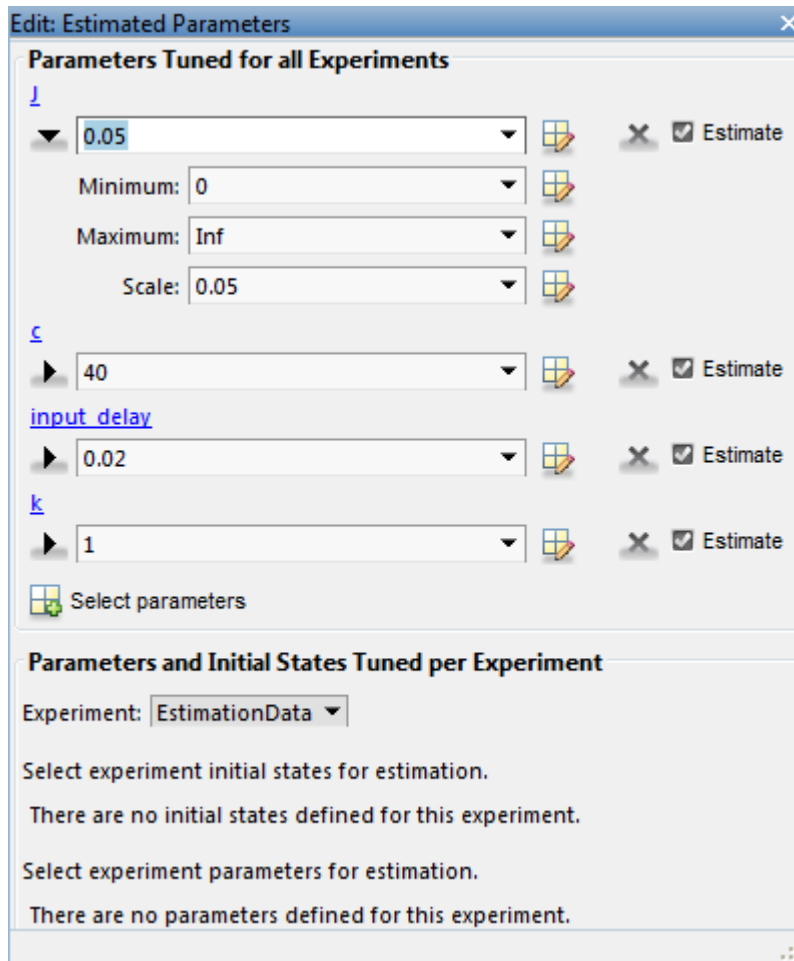
The Estimation Task

With the parameters for estimation selected we select experiments to use for estimation. Click **Select Experiments** and select `EstimationData` for estimation.
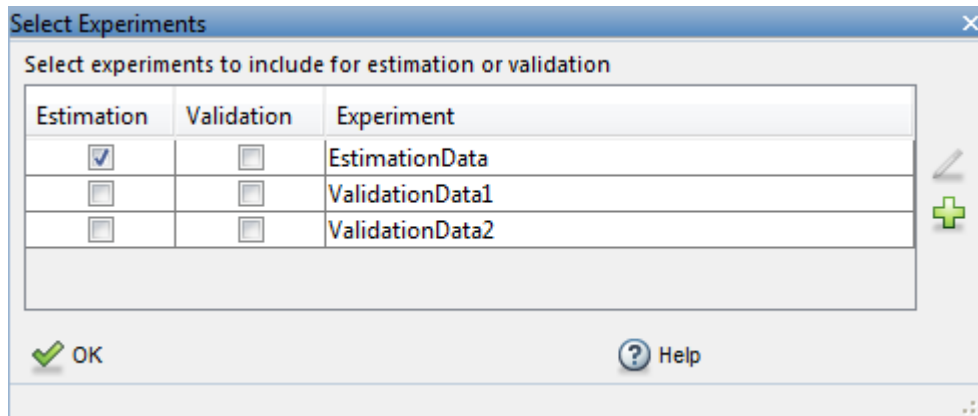
We are now almost ready to start our estimation but first create plots to monitor the estimation progress. Click **Add Plot** and select **Parameter Trajectory**. This creates a plot that shows how the estimated parameter values change during estimation. Click the **View** tab to layout the plots so that the `Experiment plot:EstimationData` and `Iteration plot 1` are both visible.
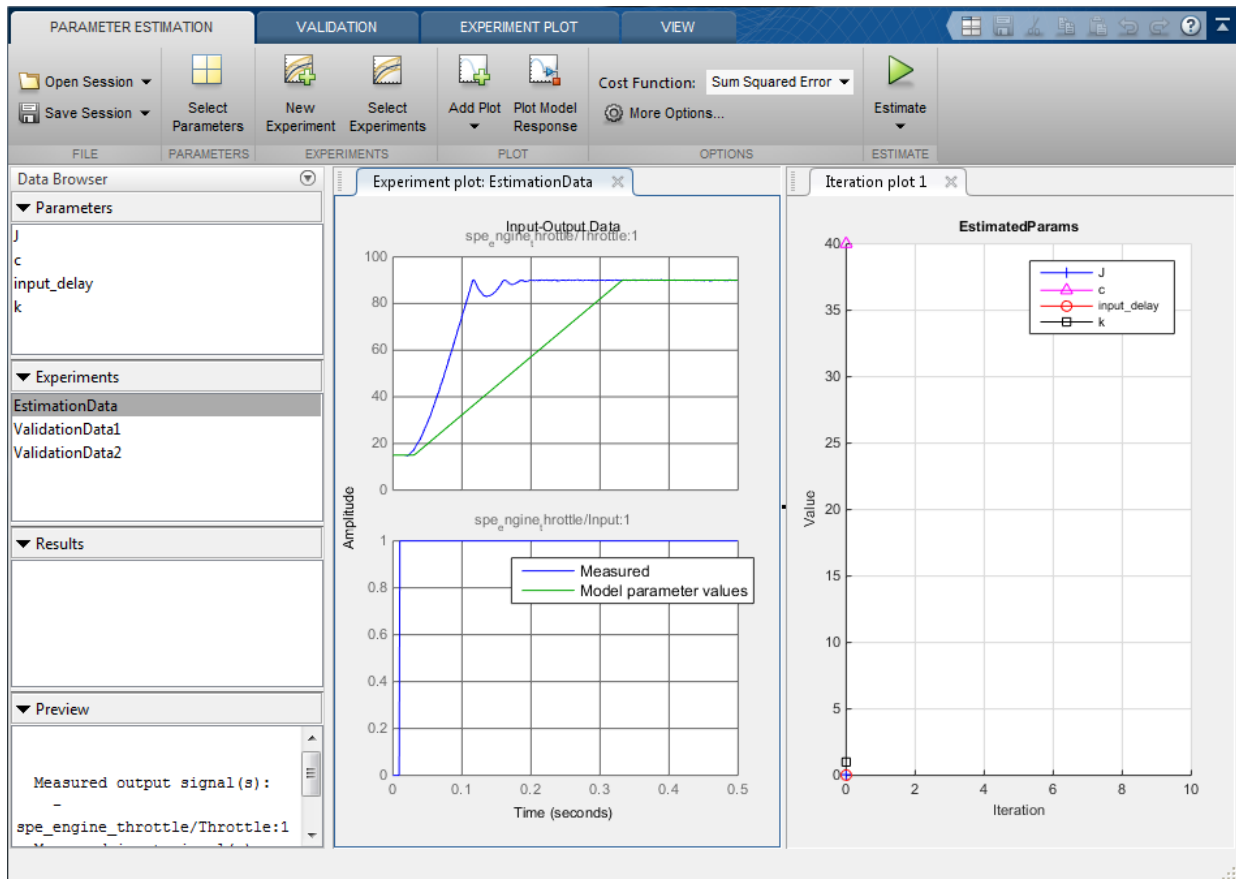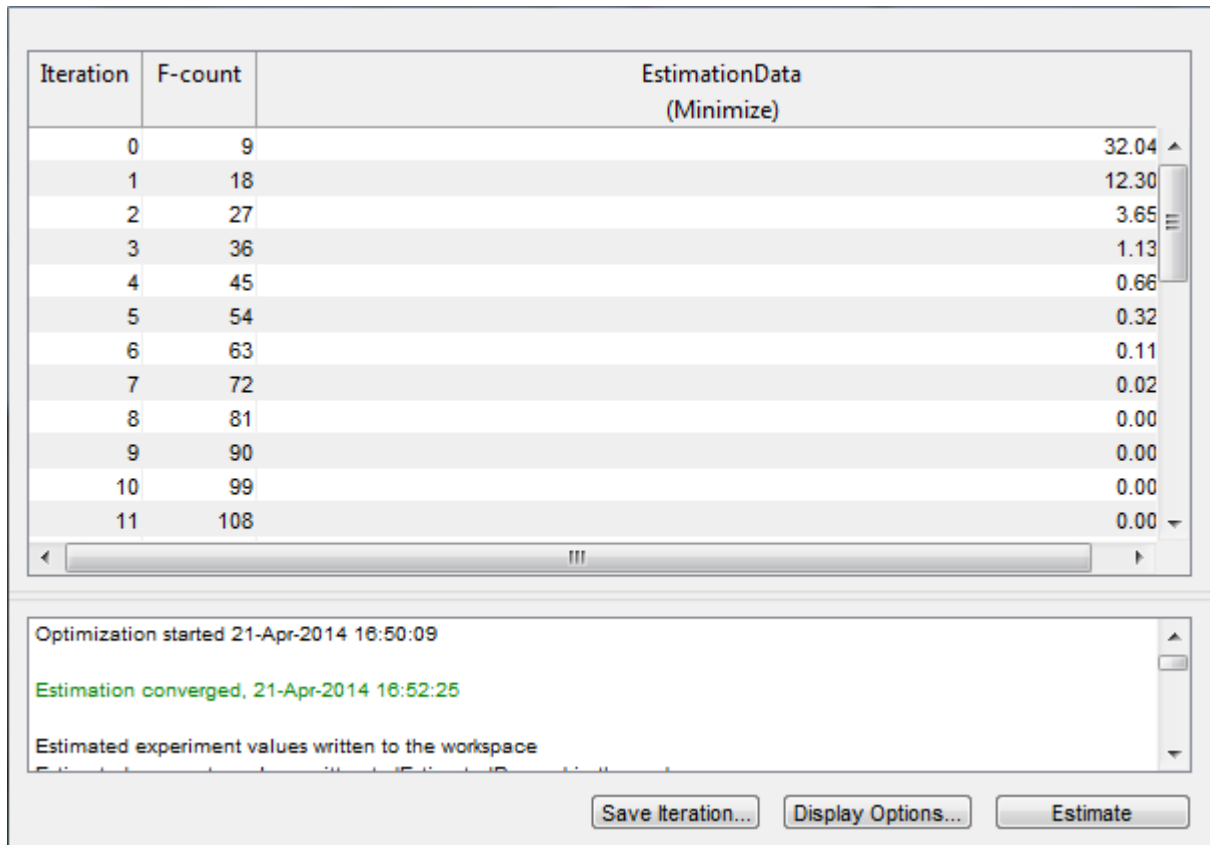
Click the **Estimate** button to start the estimation. You can modify estimation options by setting the **Cost Function** combobox and clicking **More Options...**.

While the estimation is running the plots update and a dialog showing estimation progress appears. The progress dialog shows the estimation iterations, the number of times the model has been evaluated (F-count), and the estimation cost at each iteration.

Iteration	F-count	EstimationData (Minimize)
0	9	32.04
1	18	12.30
2	27	3.65
3	36	1.13
4	45	0.66
5	54	0.32
6	63	0.11
7	72	0.02
8	81	0.00
9	90	0.00
10	99	0.00
11	108	0.00

Optimization started 21-Apr-2014 16:50:09

Estimation converged, 21-Apr-2014 16:52:25

Estimated experiment values written to the workspace

Save Iteration...   Display Options...   Estimate

After a number of iterations the estimation converges and terminates. The model is updated with the estimated parameters and the estimation results are saved in the data browser. Right click `EstimatedParams` and select **Open...** to see details of the estimation result.
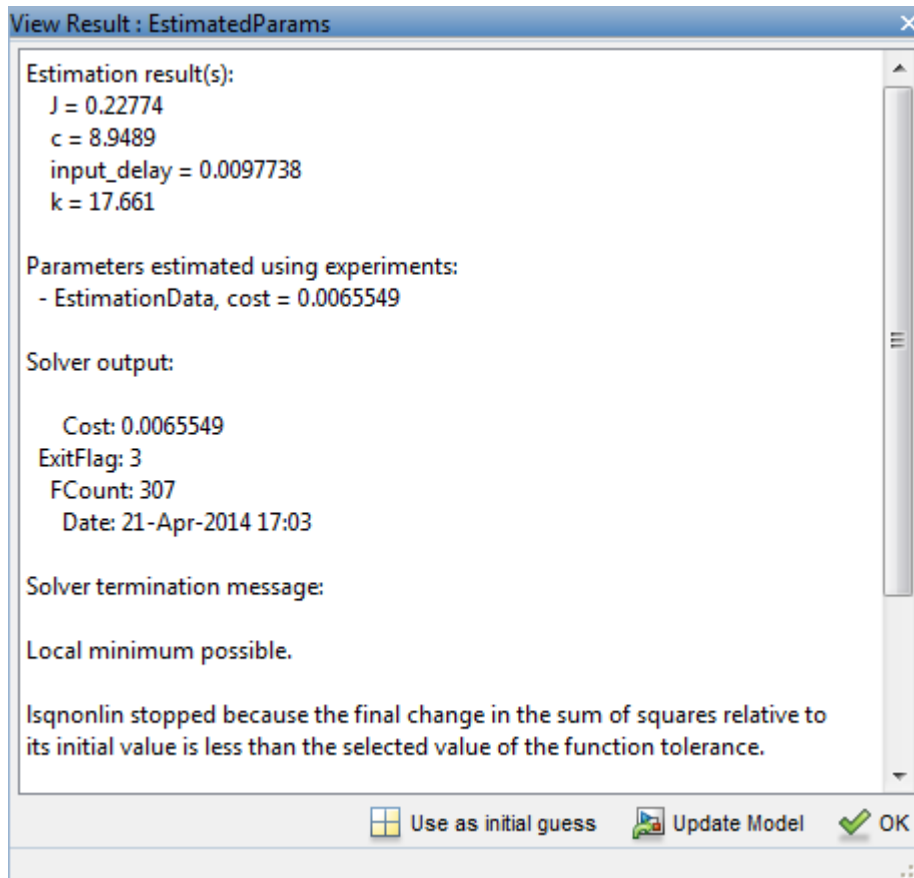
Validation

It is important to validate the estimation results against other experiments. A successful estimation will not only match the experimental data that was used for estimation but also other independent measured data that were collected in experiments.

Click the **Validation** tab and click **Select Experiments** to select experiments for validation. Select both `ValidationData1` and `ValidationData2` for validation.
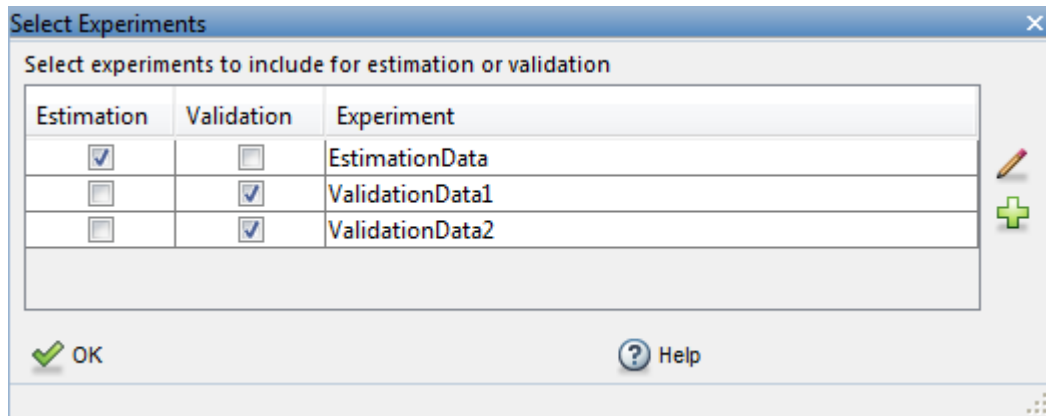
Click **Select Results** to select the estimation result(s) to use for validation. Select `EstimatedParams` and deselect `Use current parameter values`.
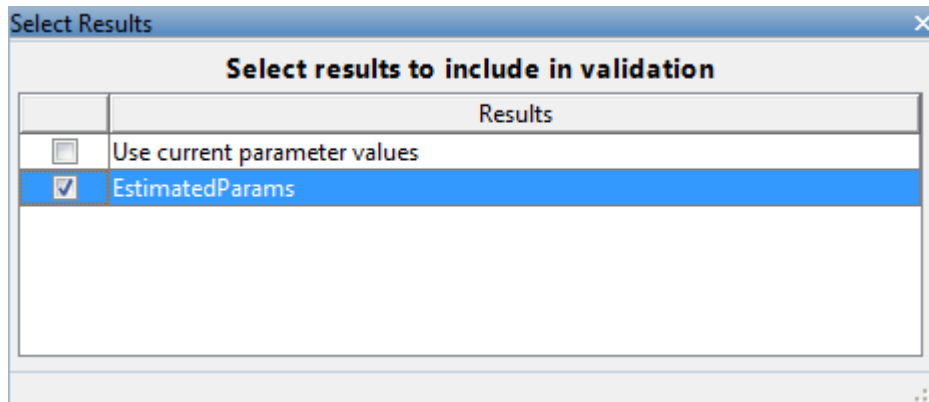
Click **Validate** to validate the estimation result against the validation experiments. Validation simulates the model using the estimated parameters and selected experiments and creates plots showing the measured and simulation data. Use the **View** tab to layout the plots so that the `Experiment plot:ValidationData1` and `Experiment plot:ValidationData2` are both visible.
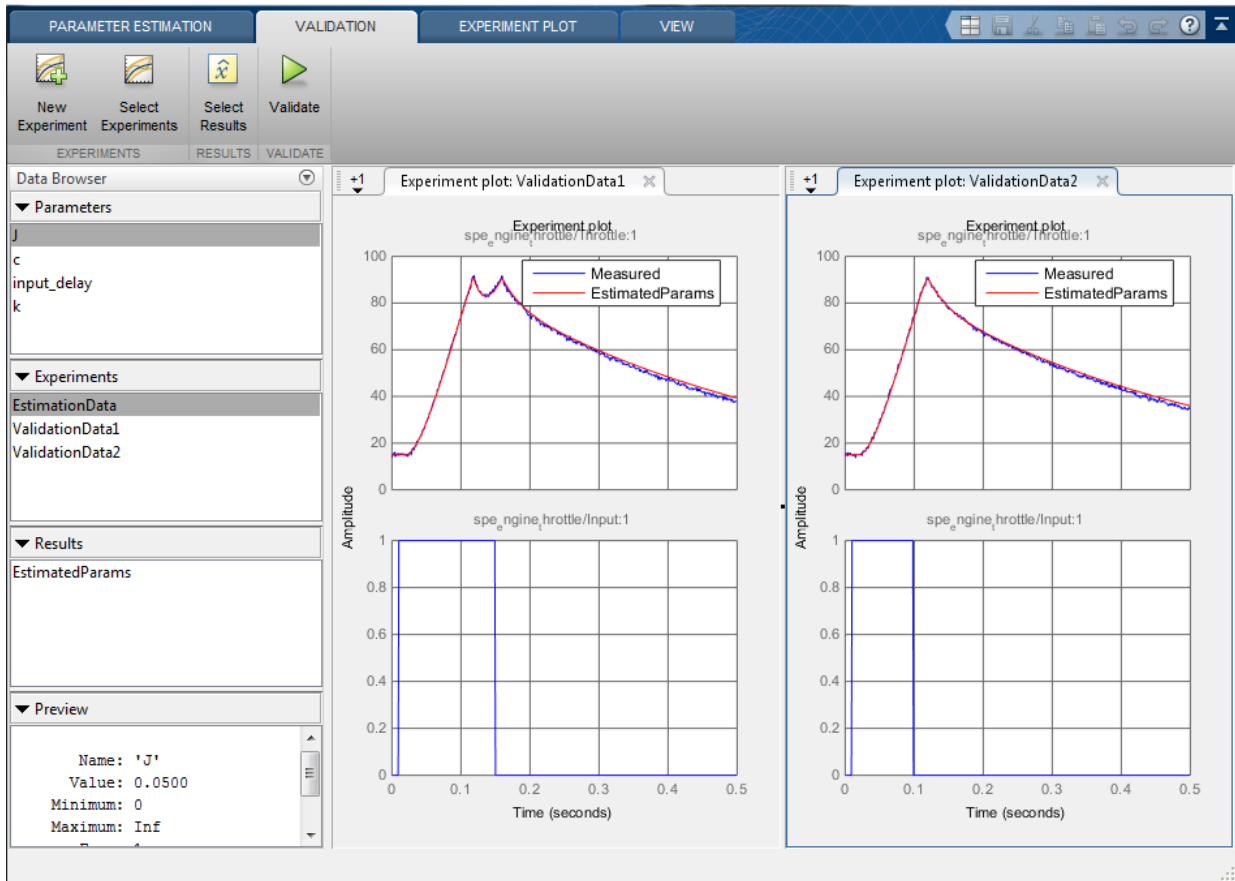
The valdiation plots confirm that our estimation was successful, showing that the estimated parameters are robust enough to handle a variety of inputs.

Related Examples

To learn how to estimate model parameters using the `sdo.optimize` command, see "Estimate Model Parameter Values (Code)".

Close the model

Estimate Model Parameters Per Experiment (GUI)

This example shows how to use multiple experiments to estimate a mix of model parameter values; some that are estimated using all the experiments and others that are estimated using individual experiments. The example also shows how to configure estimation experiments with experiment dependent parameter values.

You estimate the parameters of a rechargeable battery based on data collected in experiments that discharge and charge the battery.

Open the Model and Get Experimental Data

This example estimates parameters of a simple, rechargeable battery model, `sdoBattery`. The model input is the battery current and the model output, the battery terminal voltage, is computed from the battery state-of-charge.

```
open_system('sdoBattery');
```


Simple Battery Model

Copyright 2012-2014 The MathWorks, Inc.

The model is based on the equation

$$E = (1 - Loss) * V - K * Q_{max} * \frac{1-s}{s}$$

Where:

- E is the battery terminal voltage in Volts.

- V is the battery constant voltage in Volts.

- K is the battery polarization resistance in Ohms.

- Q_{max} is the maximum battery capacity in Ampere-Hour.

- s is the battery charge state, with 1 being fully charged and 0 zero charge. The battery state-of-charge is computed from the integral of the battery current with a +ve current indicating discharge and a -ve current indicating charging. The battery initial state-of-charge is specified by Q_0 in Ampere-Hour.

- $Loss$ is the voltage drop when charging, expressed as a fraction of the battery constant voltage. When the battery is discharging this value is zero.

`V`, `K`, `Qmax`, `Q0`, and `Loss` are variables defined in the model workspace.

Estimation Experiment Data

A 1.2V (6500mAh) battery was subjected to a discharge and a charging experiment. This experiment data has been loaded into a preconfigured estimation tool session.

From the `sdoBattery` model click **Analysis** and select the **Parameter Estimation...** menu item to launch the Parameter Estimation tool. From the Parameter Estimation tool click **Open Session** and select **Open from model workspace** and open the `sdoBattery_sdosession` session. The measured charge and discharge experiment data are loaded and plotted.

Click the **View** tab to layout the plots so that the `Experiment plot:Charge_Exp` and `Experiment plot:DCharge_Exp` are both visible. Click **Plot Model Response** to see how well the model simulation matches the measured experiment data.
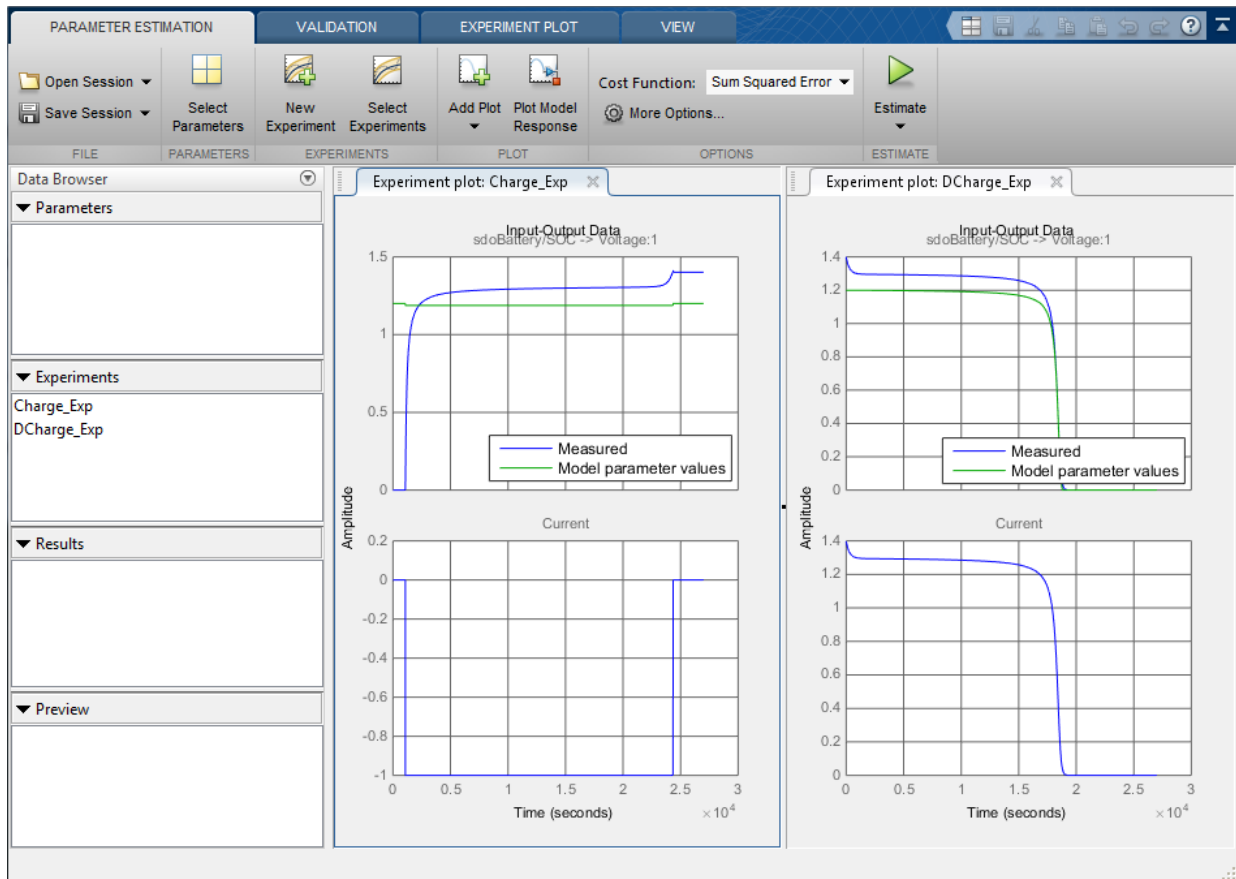
The plots show that the battery initial charge QO is not set correctly for the `Charge_Exp` experiment and that the model V, K, and `Loss` parameters need to be estimated.

Setting Experiment Parameter Values

The previous plot indicates that the `Charge_Exp` battery initial charge, QO, is not set correctly. Add the initial charge to both experiments.

Right click `Charge_Exp` and select **Edit...**. A dialog to edit the experiment opens.
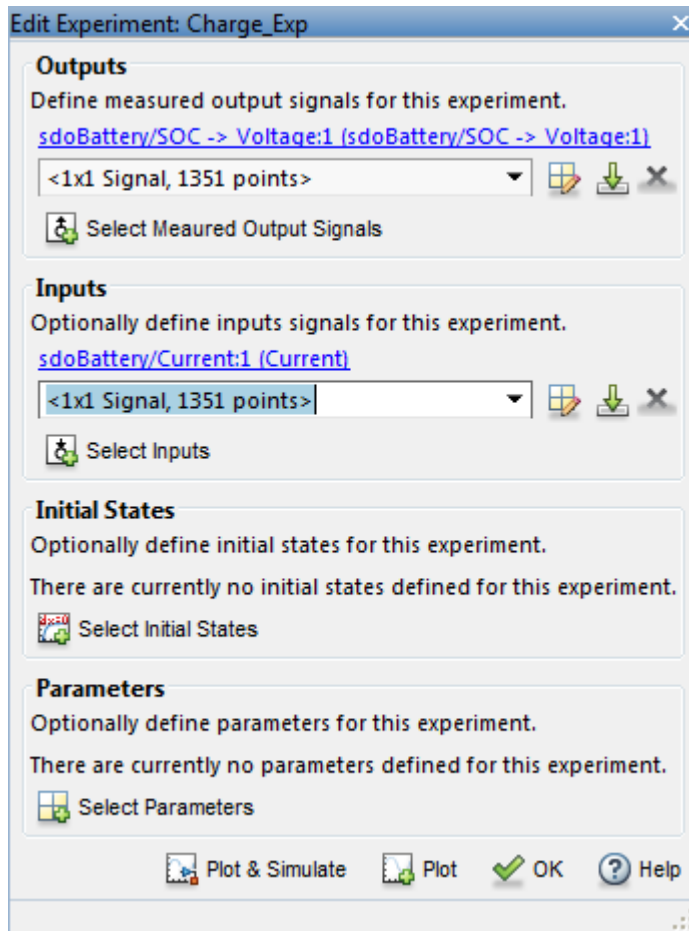
Click **Select Parameters** to open a dialog to add model parameters to the experiment. Select `Loss` and `Q0` to add to the experiment. Select `Loss` as we need to estimate this parameter using only the `Charge_Exp` experiment. Click **Ok** to add the `Q0` and `Loss` parameters to the experiment.
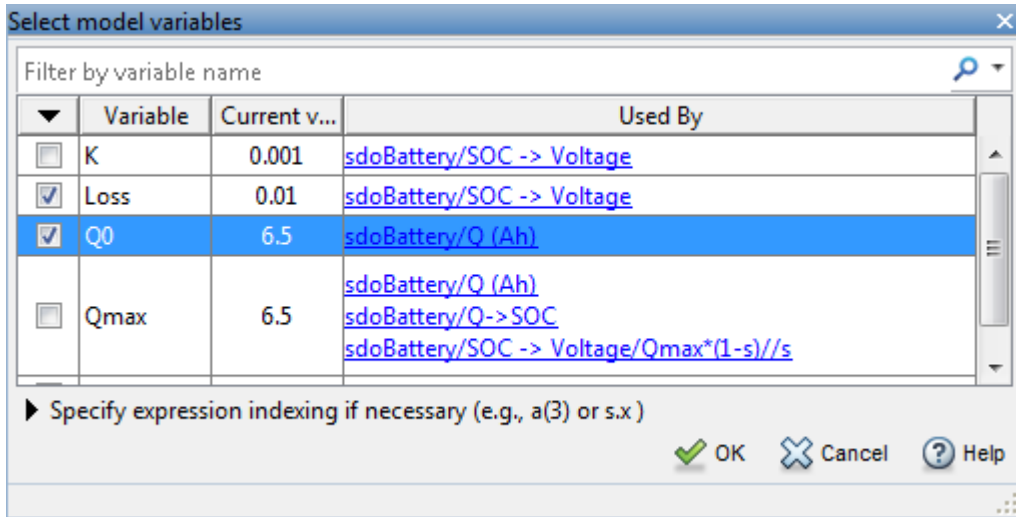
Set the battery initial charge `Q0` in the `Charge_Exp` to 0, i.e. there is no initial charge.
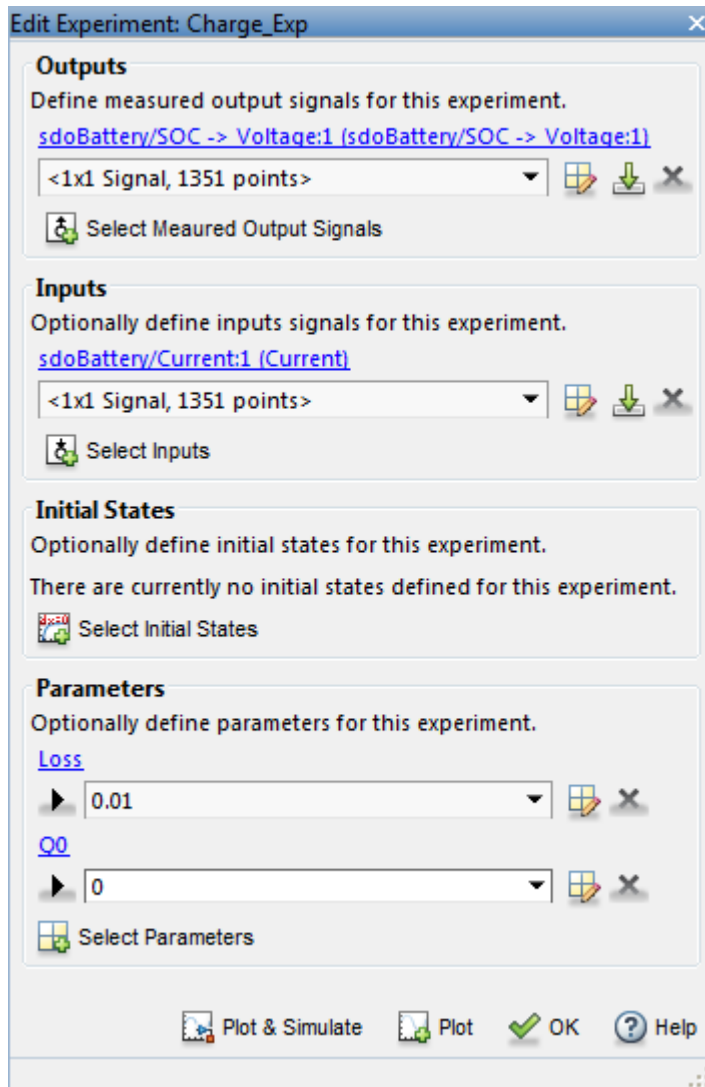
Similarly add the battery initial charge `Q0` to the `DCharge_Exp` experiment and set the initial charge to 6.5., i.e. for this experiment there is an initial charge.
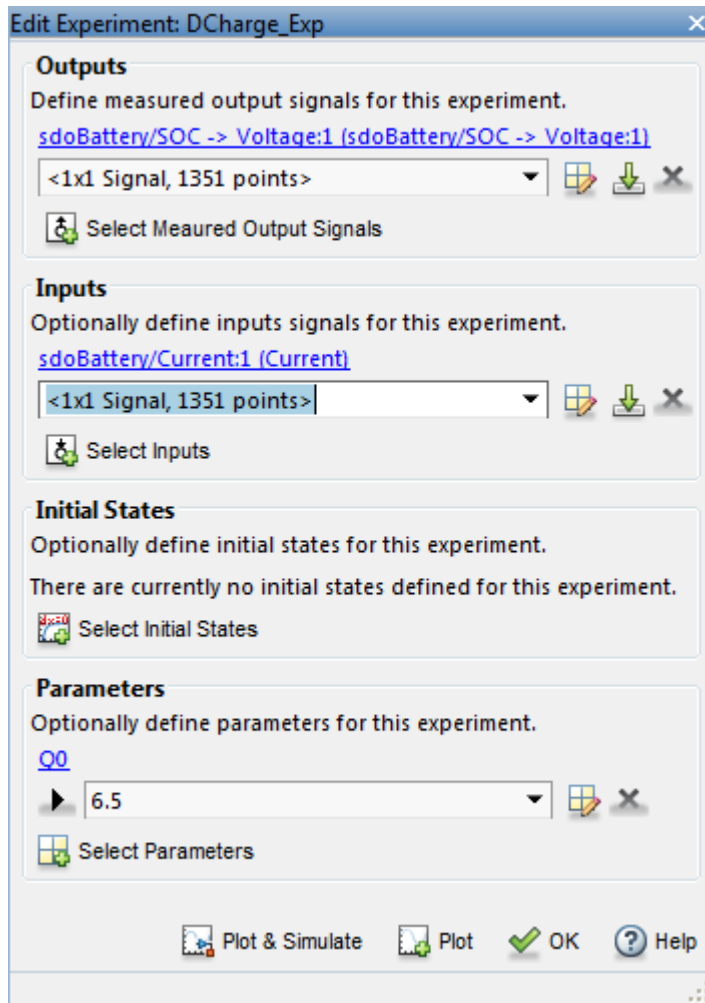
Now that the experiements are updated with the correct initial battery charge click **Plot Model Response** to simulate the model and compare measured and simuated data.
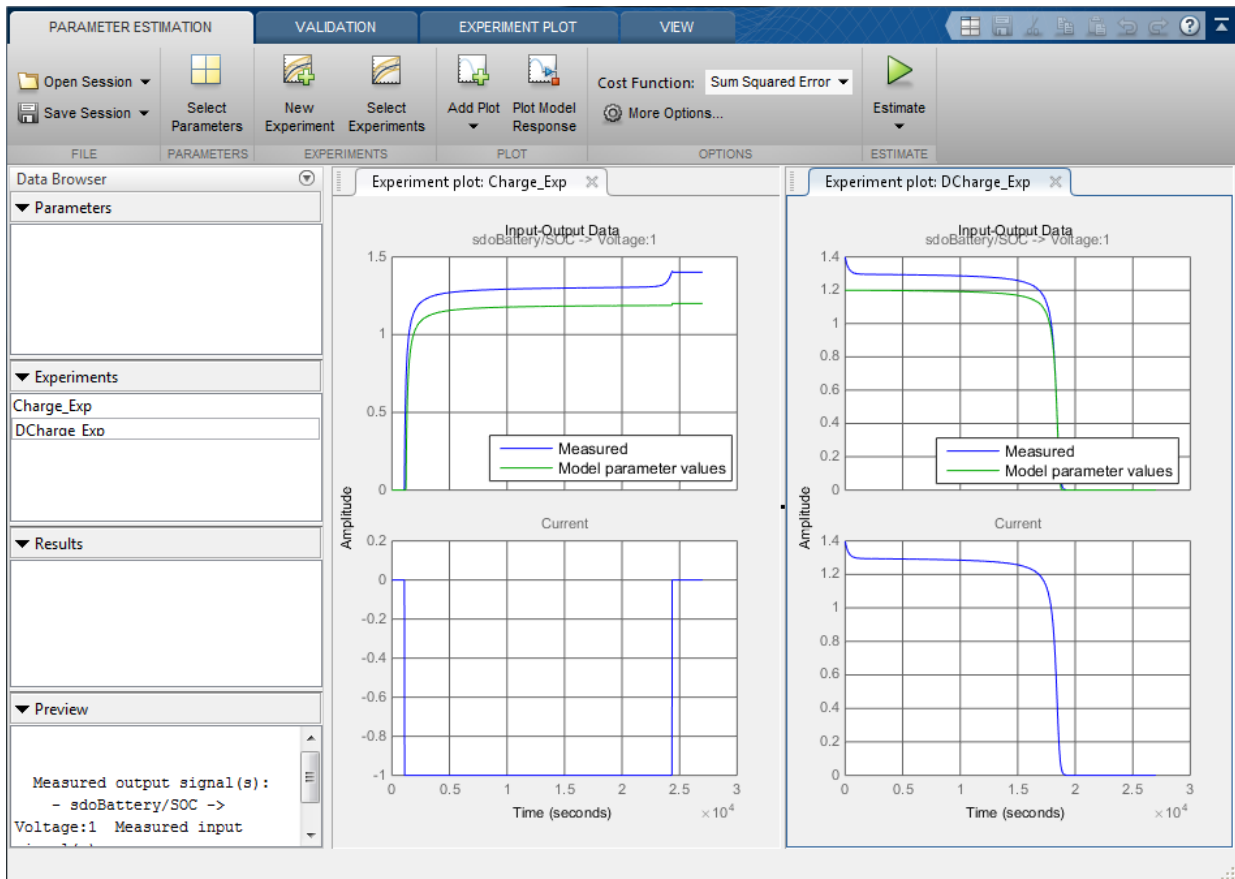
The experiment plots show that the experiment initial conditions match but the battery response does not. The next step is to estimate the K and V model parameters.

Select Estimation Parameters

The previous plot showed that the model response does not match the measured data and we need to etimate the model V and K parameters.

Click **Select Parameters** to open a dialog to select model parameters.
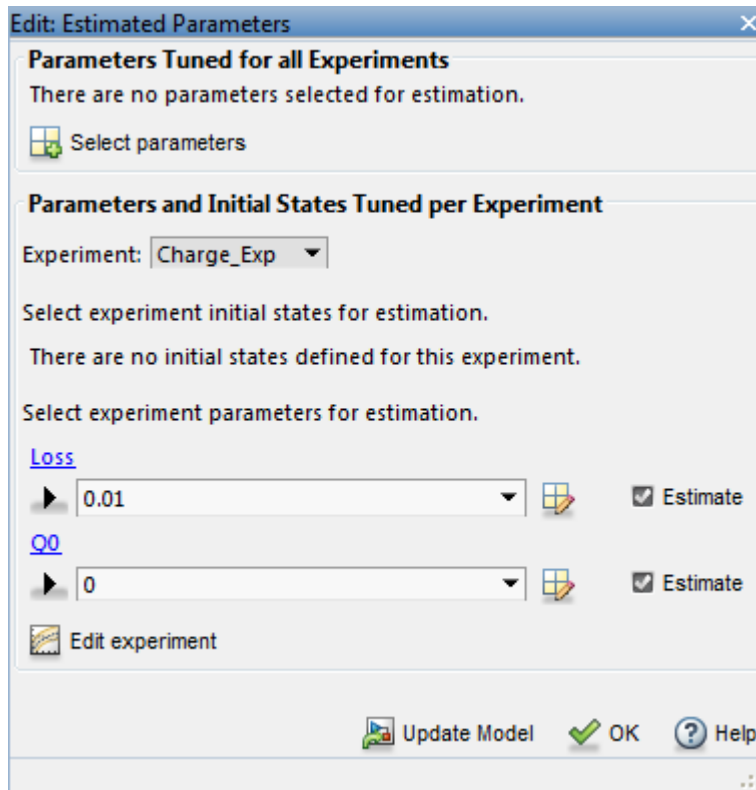
The upper portion of the select parameters dialog has a section for parameters that are tuned using all experiments. Click **Select Parameters** and add the V and K model parameters to the estimated parameters. Set the V minimum to 0 and the maximumn to 2, similarly set the K minimum to 1e-6 and maximum to 0.1.

Edit: Estimated Parameters

Parameters Tuned for all Experiments

K

0.001 ▼

Minimum: 1e-06 ▼

Maximum: 0.1 ▼

Scale: 0.001953125 ▼

☑ Estimate

V

1.2 ▼

Minimum: 0 ▼

Maximum: 2 ▼

Scale: 2 ▼

☑ Estimate

Select parameters

Parameters and Initial States Tuned per Experiment

Experiment: Charge_Exp ▼

Select experiment initial states for estimation.

There are no initial states defined for this experiment.

Select experiment parameters for estimation.

Loss

0.01 ▼   ☑ Estimate

Q0

0 ▼   ☑ Estimate

Edit experiment

Update Model   OK   Help

2-173

The lower section of the dialog has a section for initial states and parameters that are tuned using individual experiments.

For the `Charge_Exp` we tune the `Loss` parameter and set its minimum to 0 maximum to 0.5. The battery initial charge `Q0` is fixed to 0 and should not be estimated; uncheck **Estimate**.
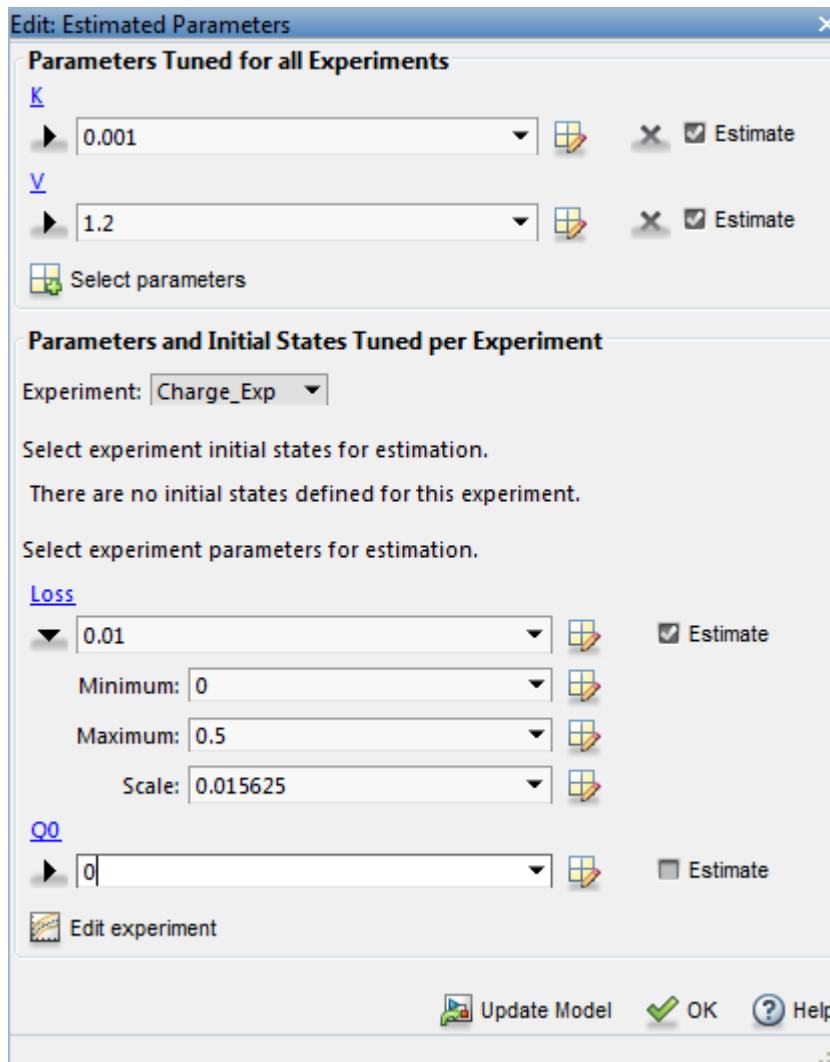
Select DCharge_Exp from the **Experiment** combobox to view the parameter settings for the DCharge_Exp experiment. The battery initial charge Q0 is fixed to 6.5 and should not be estimated; uncheck **Estimate**
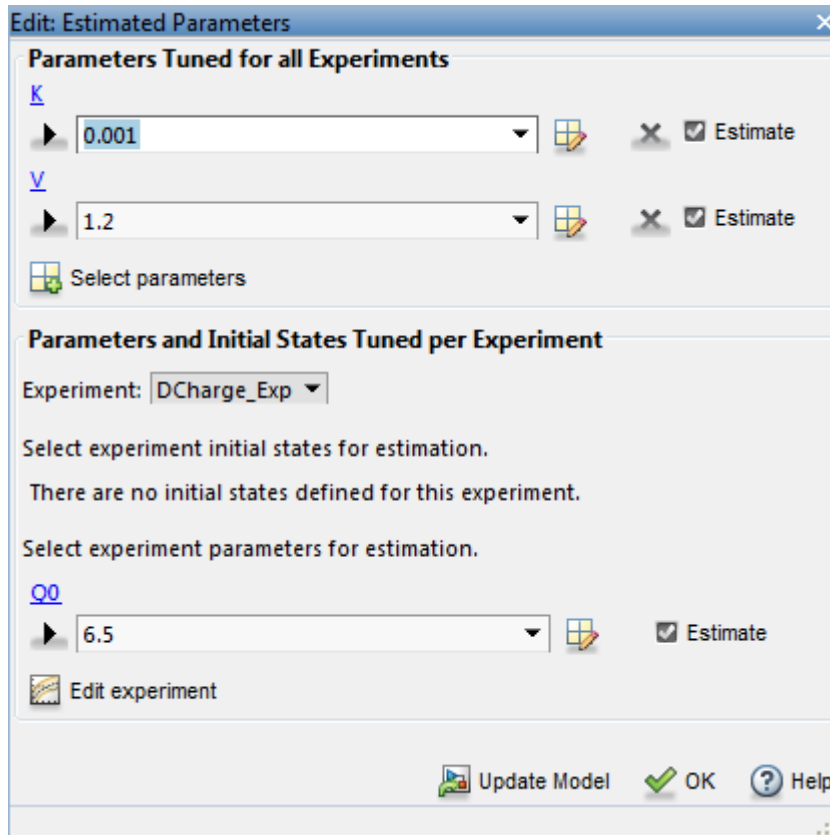
Estimate Parameter Values

The experiments and estimated parameters are configured and we a ready to run the estimation. First create a plot to monitor the estimation progress. Click **Add Plot** and select **Parameter Trajectory**. This creates a plot that shows how the estimated parameter values change during estimation. Click the **View** tab to layout the plots so that the experiment and parameter trajectory plots are all visible.
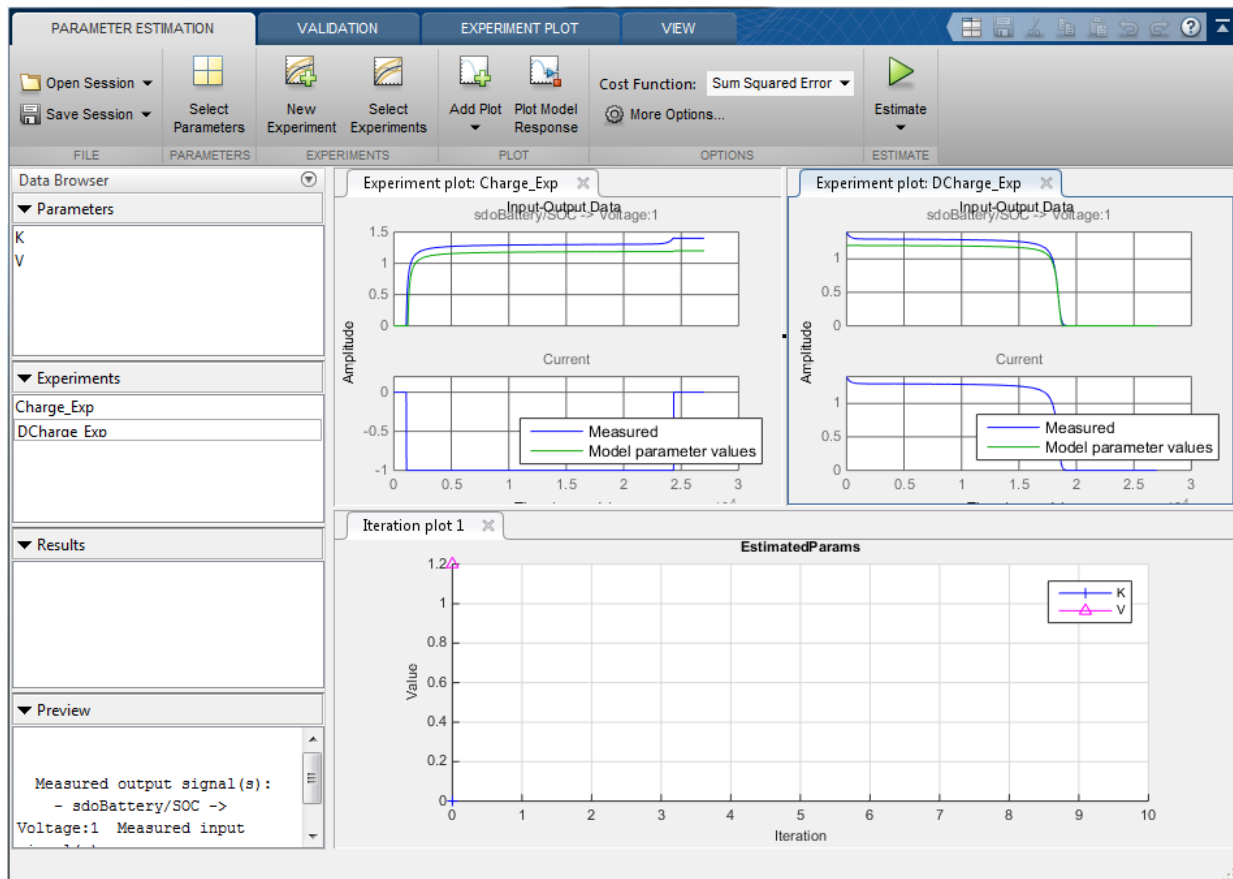
Click the **Estimate** button to start the estimation. You can modify estimation options by setting the **Cost Function** combobox and clicking **More Options...**.

While the estimation is running the plots update and a dialog showing estimation progress appears. The progress dialog shows the estimation iterations, the number of times the model has been evaluated (F-count), and the estimation cost at each iteration.
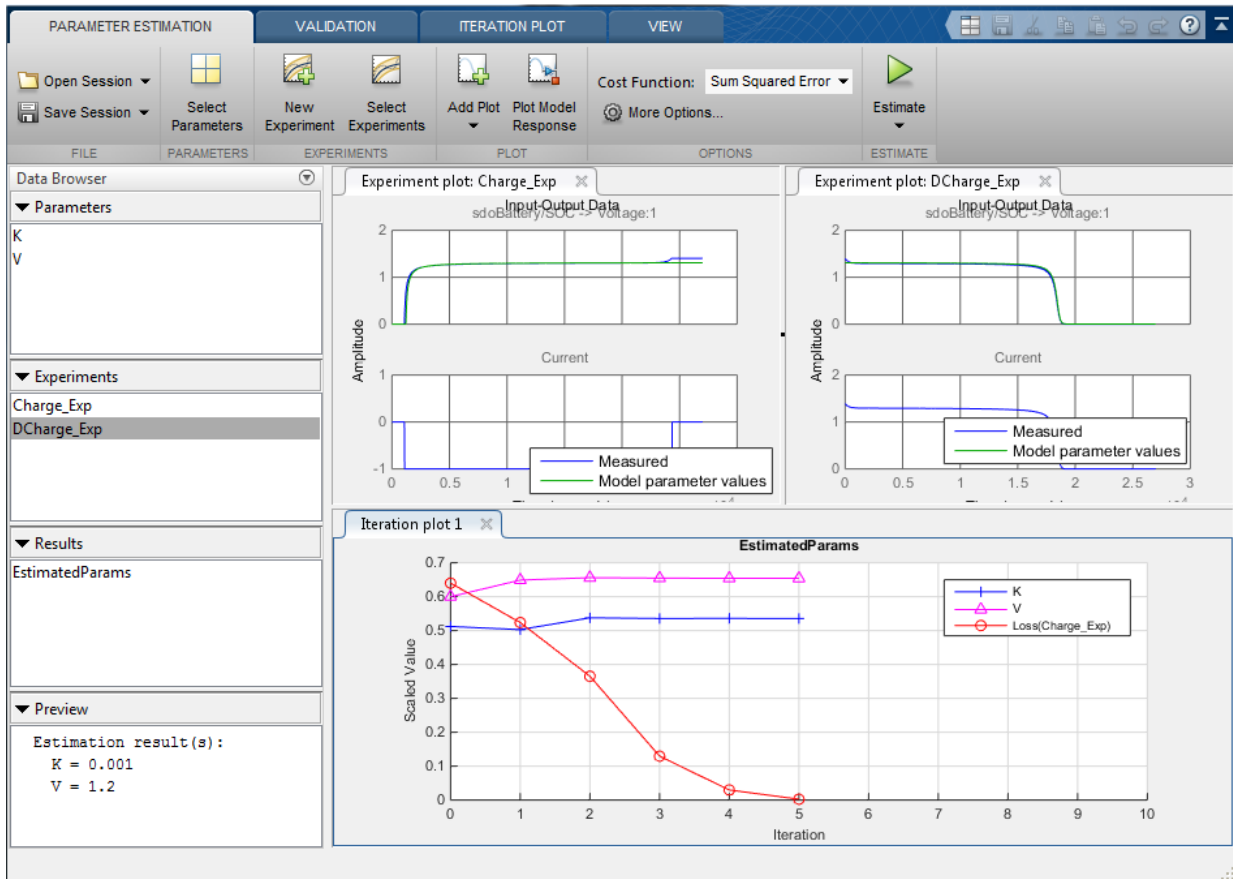
Iteration	F-count	Charge_Exp (Minimize)	DCharge_Exp (Minimize)
0	7	12.6204	4.1010
1	14	2.2115	1.3692
2	21	1.9613	0.2716
3	28	1.9663	0.2372
4	35	1.9765	0.2169
5	42	1.9756	0.2164

Optimization started 23-Apr-2014 15:10:19

Estimation converged, 23-Apr-2014 15:11:15

Estimated experiment values written to the workspace

[Save Iteration...] [Display Options...] [Estimate]

After a number of iterations the estimation converges and terminates. The experiement plots show the measured and simlation data matching well. The `EstimatedParams` plot shows the V, K, and `Loss` parameters changing during the estimation; the scale of V and K, `Loss` are different, right click on the plot and select **Show scaled values** to see how all the parameters changed from their original values.

Related Examples

To learn how to estimate parameters per expriment using the `sdo.optimize` command, see "Estimate Model Parameters Per Experiment (GUI)".

Close the model

```
bdclose('sdoBattery')
```

Estimate Model Parameters and Initial States (GUI)

This example shows how to estimate the physical parameters - mass (m), spring constant (k) and damping (b) of a simple mass-spring-damper model. This example illustrates the significance of initial state estimation.

Simulink® Model of the Mass Spring Damper System

The Simulink model for the mass-spring-damper system, `msd_system`, is shown below.

The model's output is the displacement response (position) of the mass in a mass-spring-damper system, subject to a constant force (F), and an initial displacement (x0). x0 is the initial condition of the Position integrator block. Run the simulation once to observe the response of the model to a nominal set of parameter values.
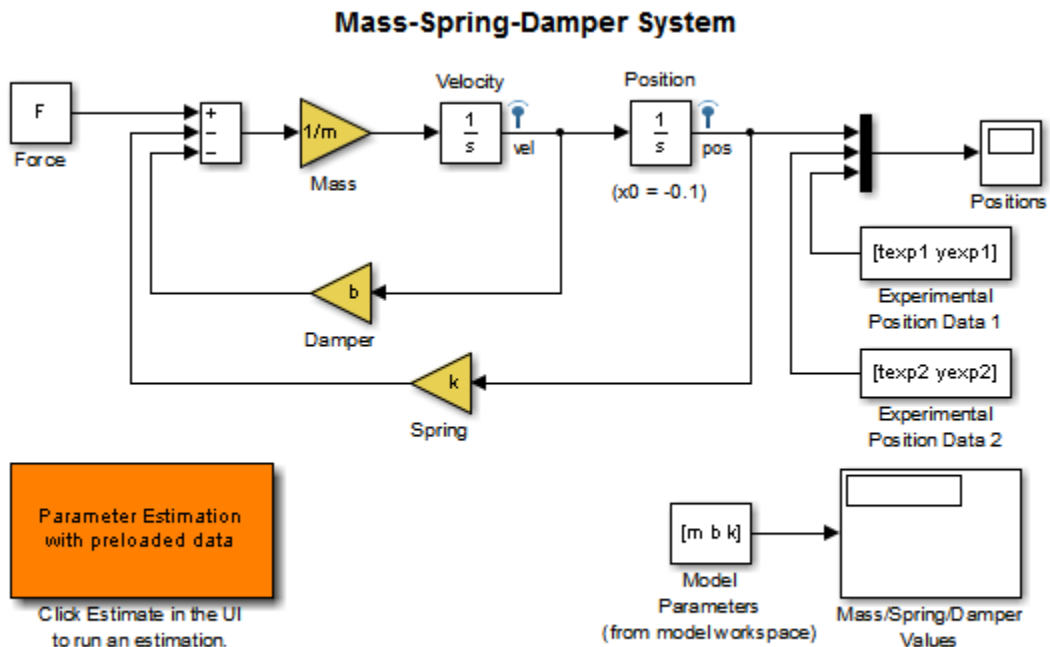
Experimental Data Sets

For estimation of the model parameters (m, b and k), two sets of experimental data is used. These data sets were obtained using two different initial positions (0.1 and 0.3), and contain additive noise. A plot of these data sets is shown below (orange and cyan curves), along with the simulated response (yellow curve) of the Simulink model for x0=-0.1 and a nominal set of parameter values (m=8, k=500, b=100).
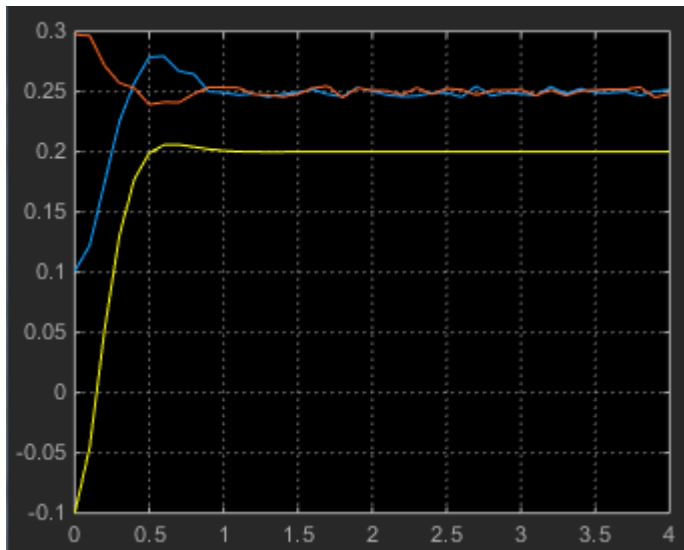
Estimation of Model Parameters

The model has three parameters (k, b, m) that appear in the Gain blocks of the Simulink model msd_system. We estimate these parameters using Parameter Estimation.

Double-click the Parameter Estimation GUI with preloaded data block in the model to open a pre-configured estimation GUI session. The experimental data sets are already loaded in the project (data_exp1 and data_exp2). Click the **View** tab to layout the plots so that the Experiment plot:data_exp1 and Experiment plot:data_exp1 are both visible. Click **Plot Model Response** to simulate the model for the two experiments. The plots show that the model simulation does not match the experiment data.
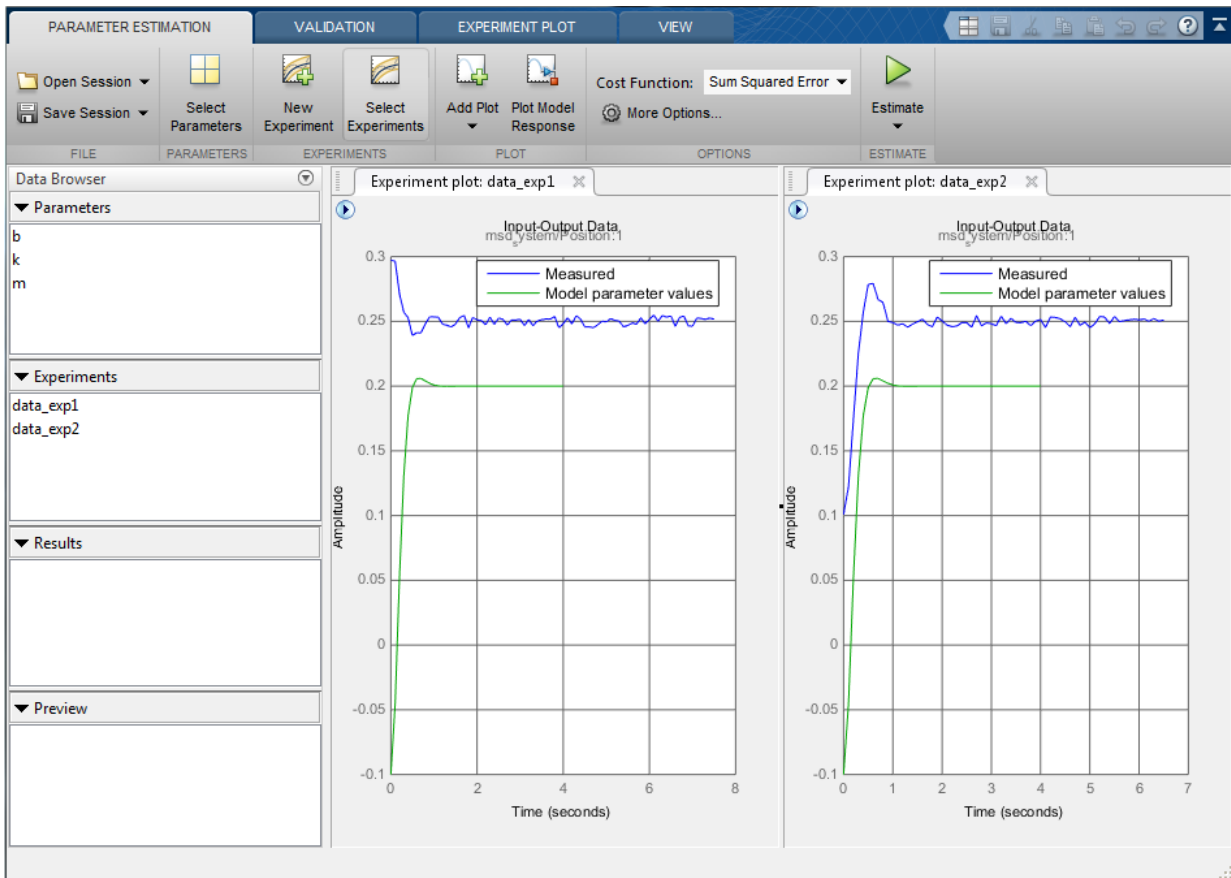
Parameter Estimation with No State Estimation

The tool has been configured to estimate the model parameters using both `data_exp1` and `data_exp2` experiments, click **Select Parameters** to see the selected parameters and **Select Experiments** to see the experiments selected for estimation.

Click **Estimate** to start the estimation. You can modify estimation options by setting the **Cost Function** combobox and clicking **More Options...**.

While the estimation is running the plots update and a dialog showing estimation progress appears. The progress dialog shows the estimation iterations, the number of times the model has been evaluated (F-count), and the estimation cost at each iteration.
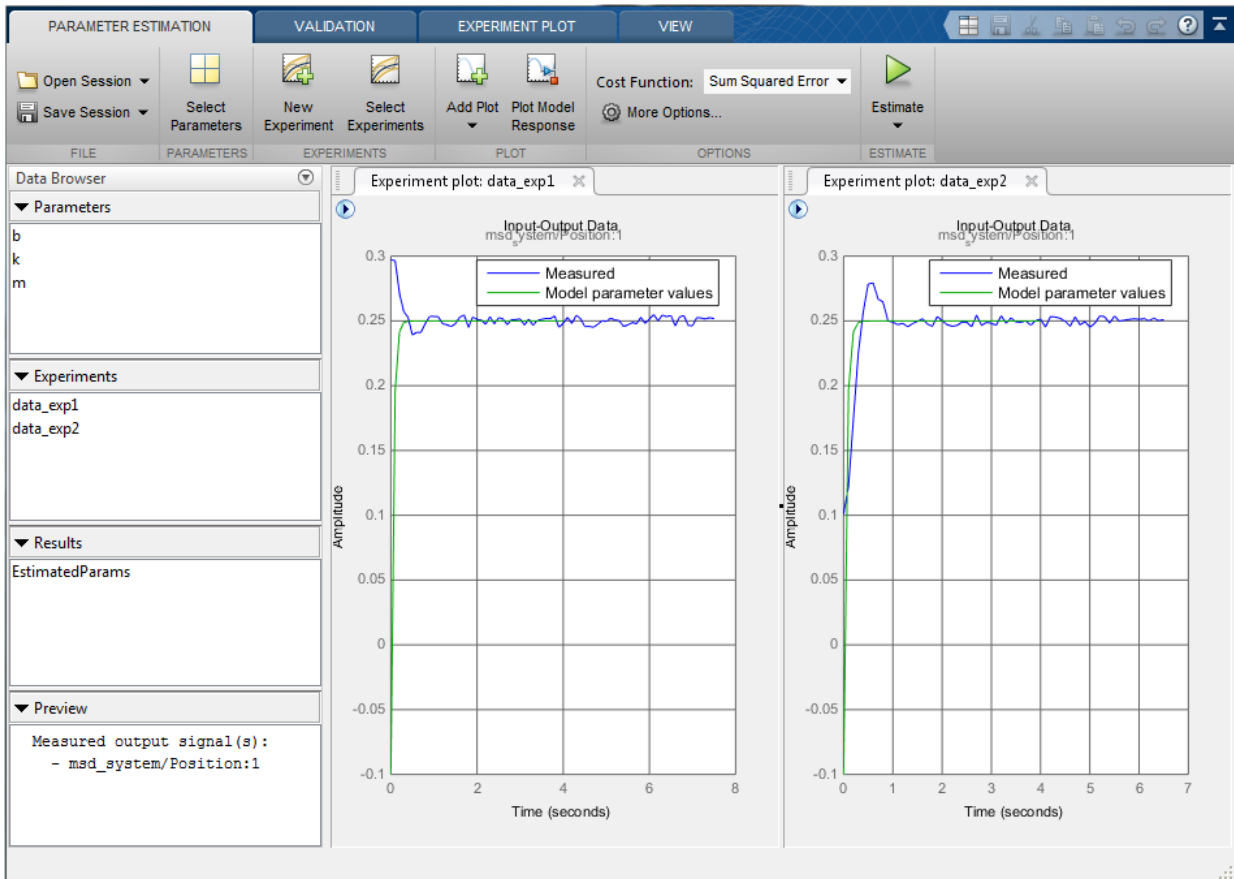
Iteration	F-count	data_exp1 (Minimize)	data_exp2 (Minimize)
0	7	4.8737	2.49
1	14	4.8737	2.49
2	21	3.7331	1.18
3	28	3.2630	0.87
4	35	3.2630	0.87
5	42	3.0230	0.74
6	49	2.8380	0.70
7	56	2.7386	0.66
8	63	2.3614	0.64
9	70	2.3129	0.56
10	77	2.0777	0.59
11	84	1.9252	0.67

Optimization started 22-Apr-2014 13:19:47

Estimation converged, 22-Apr-2014 13:21:29

Estimated experiment values written to the workspace

[Save Iteration...] [Display Options...] [Estimate]

After a number of iterations the estimation converges and terminates. The model is updated with the estimated parameters and the estimation results are saved in the data browser.

The data_exp1 and data_exp2 experiment plots show that the model parameters have been tuned to match the measured experiment data as closely as possible. The simulated measured signals match well from the 2 second mark onward but don't match well before 2 seconds. The simulation results for both experiments start at -0.1. This is the initial condition of the model which was not estimated; these plots show that the initial condition should also be estimated.

Parameter Estimation with Initial State Estimation

The `data_exp1` and `data_exp2` experiments specify the measured output data but as seen above must also specify the model initial state. We now add the initial states to the experiments and estimate them.

Right click `data_exp1` and select **Edit...** to open a dialog to configure the experiment.
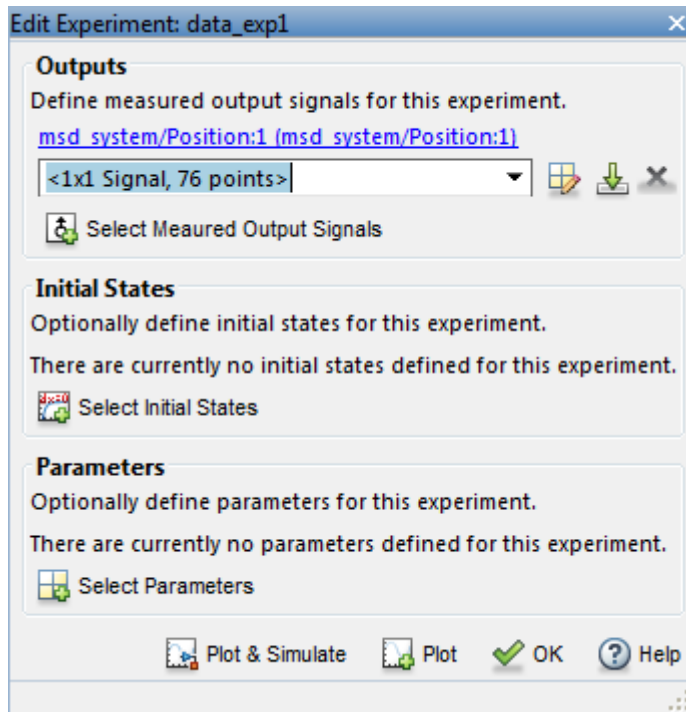
Click **Select Initial States** and select the position state. Click **OK** to close the state selector and add the selected state to the experiment.

Select model states

Filter by state name		
▼	**State**	**Current value**
☑	msd_system/Position	-0.1
☐	msd_system/Velocity	0

✔ OK   ✕ Cancel   ? Help
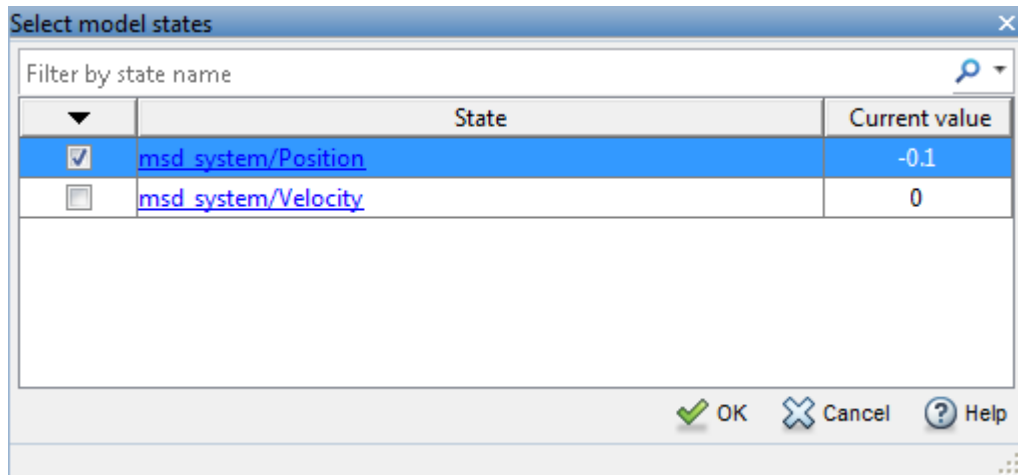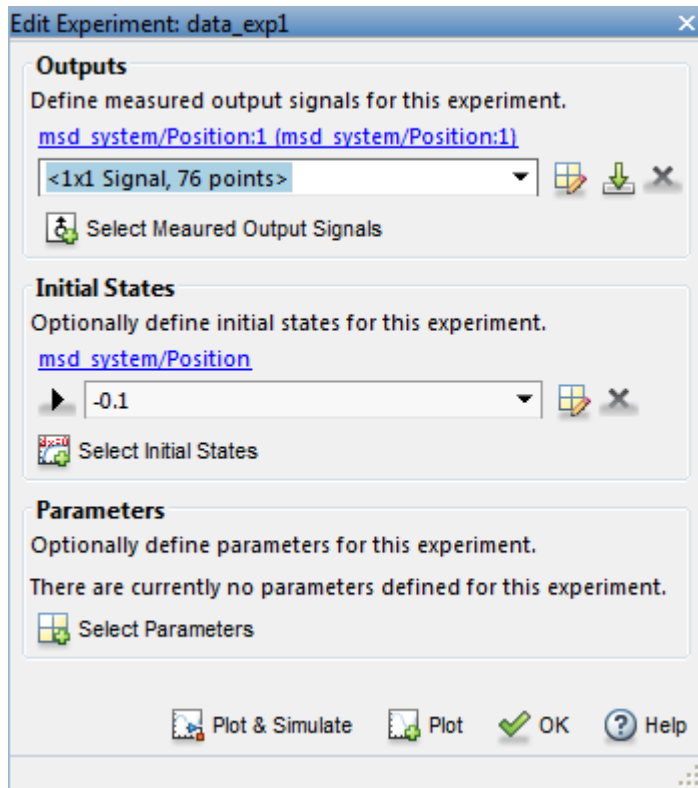
Right click `data_exp2` and select **Edit..** and add the position state to the experiment.

The experiments are now configured to include initial states that can be estimated. Click **Select Parameters**.
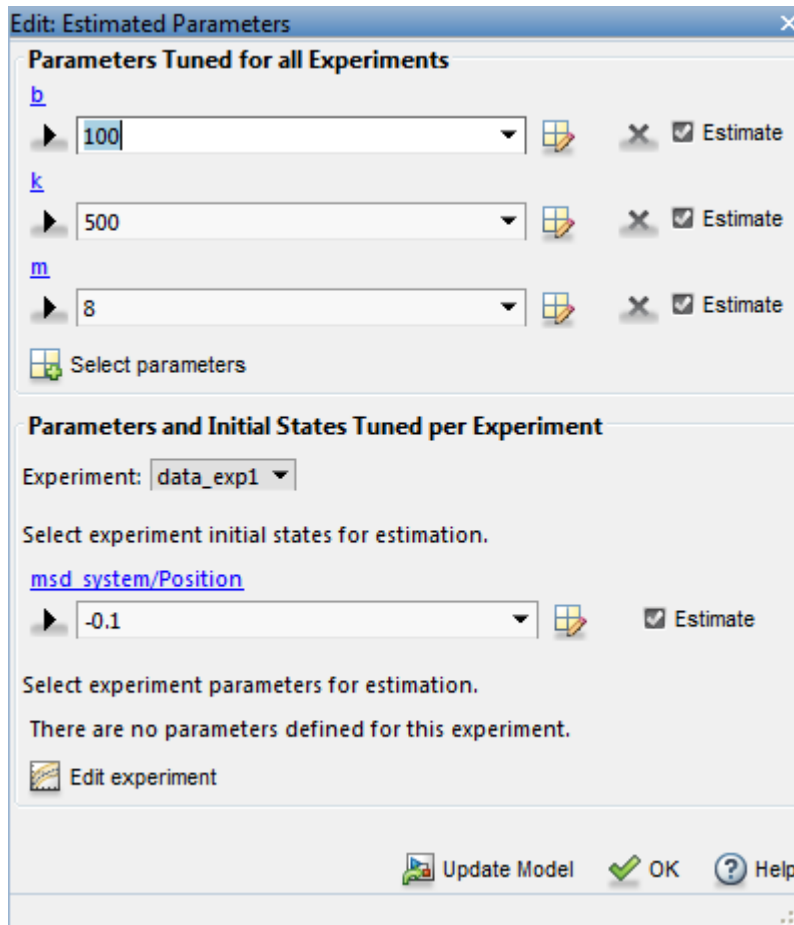
The upper portion of the select parameters dialog has a section for parameters that are tuned using all experiments selected for estimation. The lower section of the dialog has a combo-box to select an experiment and widgets to specify initial states and parameters that are tuned using only the selected experiment. For this problem the `data_exp1` and `data_exp2` experiments estimate the model intial state for each experiment.

Now we are ready to start our estimation but first create plots to monitor the estimation progress. Click **Add Plot** and select **Parameter Trajectory**, right click the plot and select **Show scaled values**. This creates a plot that shows how the estimated parameter values change during estimation. Click the **View** tab to layout the plots so that the

Experiment plot:data_exp1, Experiment plot:data_exp2, and Iteration plot 1 are both visible.

Click the **Estimate** button to start the estimation.
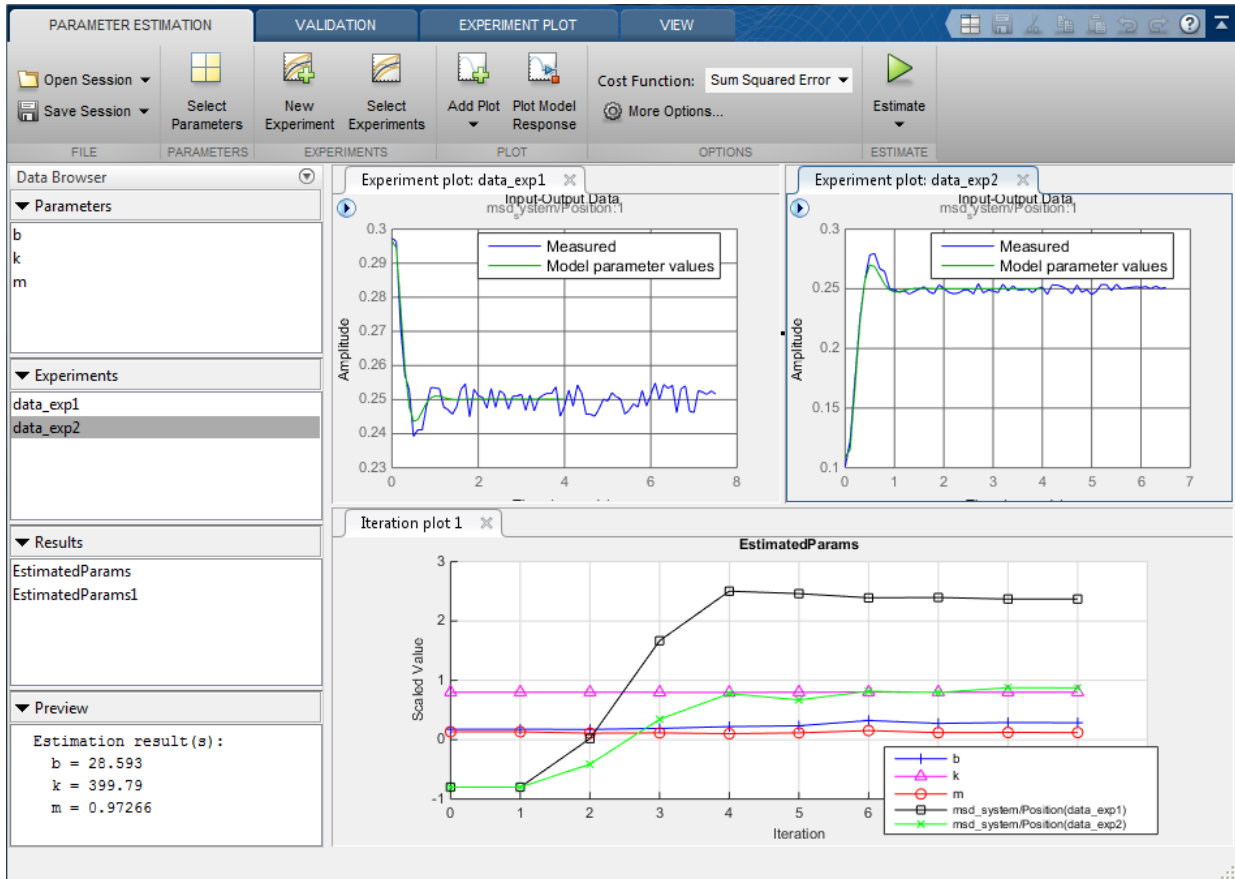
After a number of iterations the estimation converges and terminates. The data_exp1 and data_exp2 experiment plots show how estimating the initial value inproves the estimation fit. The EstimatedParams plot shows the estimated initial state for the two experiments, the plot also shows that the estimated k value did not change while b and m changed slightly. You can confirm this by clicking EstimatedParams and examining the preview pane and then clicking EstimatedParams1 and examining the preview pane.

Alternatively right click `EstimatedParams` and select **Open...** to open a dialog to view the results.

▼ Results	▼ Results
EstimatedParams	EstimatedParams
EstimatedParams1	EstimatedParams1

▼ Preview	▼ Preview

Estimation result(s):
    b = 17.559
    k = 400.07
    m = 1.0676
	Estimation result(s):
	    b = 28.593
	    k = 399.79
	    m = 0.97266

This example shows that it is important to independently estimate initial states for each experiment in order to obtain the correct estimates of the model parameters.

Related Examples

To learn how to estimate model parameters and initial states using the `sdo.optimize` command, see "Estimate Model Parameters and Initial States (Code)".

Close the model

Generate MATLAB Code for Parameter Estimation Problems (GUI)

This example shows how to automatically generate a MATLAB function to solve a Parameter Estimation problem. You use the Parameter Estimation tool to define an estimation problem for a mass-spring-damper and generate MATLAB code to solve this estimation problem.

Mass-Spring-Damper Estimation Problem

The "Estimate Model Parameters and Initial States" example shows how to use the Parameter Estimation tool to estimate parameters of a mass-spring-damper model. In this example we load a pre-configured Parameter Estimation tool session based on that example.

```
load sdoMassSpringDamper_sdosession
spetool(SDOSessionData)
```


Copyright 2002-2014 The MathWorks, Inc.

Generate MATLAB Code

From the **Estimate** list, select **Generate MATLAB Code**.

The generated code is added to the MATLAB editor as an unsaved MATLAB function.

Examine the generated code. Significant code portions are:

- **Specify Model Parameters to Estimate** - Definition of the model parameters being estimated.

- **Define the Estimation Experiments** - Definition of the measured and expected signal dat to use for estimation.

- **Create Estimation Objective Function** - Creation of an anonymous function that calls the subfunction `msd_system_optFcn`, which evaluates the model using each experiment and compares simulation and measured experiment outputs. This anonymous function is called by `sdo.optimize` at each iteration of the optimiztion problem to solve the estimation problem.

- **Estimate the Parameters** - Solve the estimation prioblem using the `sdo.optimize` command.

Select **Save** from the MATLAB editor to save the generated function.

Run Generated Code

Run the generated function.

```
Command Window
New to MATLAB? See resources for Getting Started.

>> [pEst,estInfo] = spe_msd_system;

 Optimization started 22-Oct-2014 07:42:36

                              Step-size  First-order
 Iter F-count         f(x)               optimality
    0     11       9.18411          1
    1     22     0.0502487       3.697       0.0807
    2     33     0.0140152      0.3313       0.0262
    3     44     0.0132182     0.06352      0.00053
Local minimum found.

Optimization completed because the size of the gradient is less than
the selected value of the function tolerance.
fx >>
```

The first output argument, `pOpt`, contains the optimized parameter values and the second output argument, `optInfo`, contains optimization information.

Modify the Generated Code

You can:

- Modify the generated `spe_msd_system` function to include or exclude new experiments or change estimation options.

- Call the generated `spe_msd_system` function with a different set of parameters to estimate.

For details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

Close the model

```
delete(sdotool('msd_system','ParameterEstimation'))
bdclose('msd_system')
```

3

Response Optimization

How the Optimization Algorithm Formulates Minimization Problems

When you optimize parameters of a Simulink model to meet design requirements, Simulink Design Optimization software automatically converts the requirements into a constrained optimization problem and then solves the problem using optimization techniques. The constrained optimization problem iteratively simulates the Simulink model, compares the results of the simulations with the constraint objectives, and uses optimization methods to adjust tuned parameters to better meet the objectives.

This topic describes how the software formulates the constrained optimization problem used by the optimization algorithms. For each optimization algorithm, the software formulates one of the following types of minimization problems:

- Feasibility
- Tracking
- Mixed feasibility and tracking

For more information on how each optimization algorithm formulates these problems, see:

- "Gradient Descent Method Problem Formulations" on page 3-7
- "Simplex Search Method Problem Formulations" on page 3-8
- "Pattern Search Method Problem Formulations" on page 3-9
- "Gradient Computations" on page 3-10

Feasibility Problem and Constraint Formulation

Feasibility means that the optimization algorithm finds parameter values that satisfy all constraints to within specified tolerances but does not minimize any objective or cost function in doing so.

In the following figure, x_1, x_3, and x_n represent a combination of parameter values P_1 and P_2 and are feasible solutions because they do not violate the lower bound constraint.
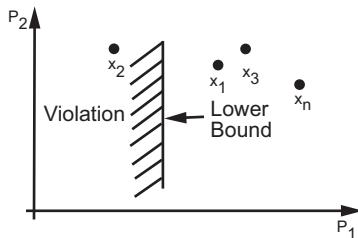
In a Simulink model, you constrain a signal by specifying lower and upper bounds in a Check block (Check Step Response Characteristics, ...) or a requirement object (`sdo.requirements.StepResponseEnvelope`, ...), as shown in the following figure.
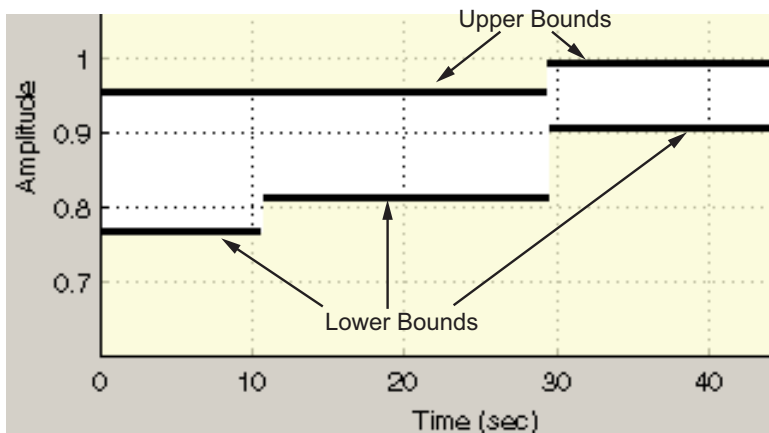
These constraints are *piecewise linear bounds*. A piecewise linear bound y_{bnd} with n edges can be represented as:

$$y_{bnd}(t) = \begin{cases} y_1(t) & t_1 \leq t \leq t_2 \\ y_2(t) & t_2 \leq t \leq t_3 \\ \vdots & \vdots \\ y_n(t) & t_n \leq t \leq t_{n+1} \end{cases},$$

The software computes the signed distance between the simulated response and the edge. The signed distance for lower bounds is:

$$c = \begin{bmatrix} \max\limits_{t_1 \leq t \leq t_2} y_{bnd} - y_{sim} \\ \max\limits_{t_2 \leq t \leq t_3} y_{bnd} - y_{sim} \\ \max\limits_{t_n \leq t \leq t_{n+1}} y_{bnd} - y_{sim} \end{bmatrix},$$

where y_{sim} is the simulated response and is a function of the parameters being optimized.

The signed distance for upper bounds is:

$$c = \begin{bmatrix} \max\limits_{t_1 \leq t \leq t_2} y_{sim} - y_{bnd} \\ \max\limits_{t_2 \leq t \leq t_3} y_{sim} - y_{bnd} \\ \max\limits_{t_n \leq t \leq t_{n+1}} y_{sim} - y_{bnd} \end{bmatrix}.$$

At the command line, `opt_fcn` supplies c directly from the `Cleq` field of `vals`.

If *all* the constraints are met ($c \leq 0$) for some combination of parameter values, then that solution is said to be feasible. In the following figure, x_1 and x_3 are feasible solutions.
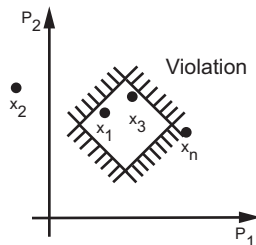
When your model has multiple requirements or vector signals feeding a requirement, the constraint vector is extended with the constraint violations for each signal and bound:

$$C = [c_1; c_2; \cdots; c_n].$$

Tracking Problem

In addition to lower and upper bounds, you can specify a reference signal in a Check Against Reference block or `sdo.requirements.SignalTracking` object, which the Simulink model output can track. The tracking objective is a sum-squared-error tracking objective.

You specify the reference signal as a sequence of time-amplitude pairs:

$$y_{ref}(t_{ref}), \ t_{ref} \in \{T_{ref0}, T_{ref1}, \cdots, T_{refN}\}.$$

The software computes the simulated response as a sequence of time-amplitude pairs:

$$y_{sim}(t_{sim}), \ t_{sim} \in \{T_{sim0}, T_{sim1}, \cdots, T_{simN}\},$$

where some values of t_{sim} may match the values of t_{ref}.

A new time base, t_{new}, is formed from the union of the elements of t_{ref} and t_{sim}. Elements that are not within the minimum-maximum range of both t_{ref} and t_{sim} are omitted:

$$t_{new} = \{t : t_{sim} \cup t_{ref}\}$$

Using linear interpolation, the software computes the values of y_{ref} and y_{sim} at the time points in t_{new} and then computes the scaled error:

$$e(t_{new}) = \frac{\left(y_{sim}(t_{new}) - y_{ref}(t_{new})\right)}{\max\limits_{t_{new}} \left|y_{ref}\right|}.$$

Finally, the software computes the weighted, integral square error:

$$f = \int w(t) e(t)^2 \, dt.$$

Note: The weight $w(t)$ is 1 by default. You can specify a different value of weight only at the command line.

When your model has requirements or vector signals feeding a requirement, the tracking objective equals the sum of the individual tracking integral errors for each signal:

$$F = \sum f_i.$$

Gradient Descent Method Problem Formulations

The Gradient Descent method uses the Optimization Toolbox function `fmincon` to optimize model parameters to meet design requirements.

Problem Type	Problem Formulation
Feasibility Problem	The software formulates the constraint $C(x)$ as described in "Feasibility Problem and Constraint Formulation" on page 3-3.
	• If you select the maximally feasible solution option (i.e., the optimization continues after an initial feasible solution is found), the software uses the following problem formulation:
	$$\begin{aligned} &\min_{[x,\gamma]} \ \gamma \\ &s.t. \quad C(x) \leq \gamma \\ &\qquad \underline{x} \leq x \leq \bar{x} \\ &\qquad \gamma \leq 0 \end{aligned}$$
	γ is a slack variable that permits a feasible solution with $C(x) \leq \gamma$ rather than $C(x) \leq 0$.
	• If you do not select the maximally feasible solution option (i.e., the optimization terminates as soon as a feasible solution is found), the software uses the following problem formulation:
	$$\begin{aligned} &\min_{x} \ 0 \\ &s.t. \quad C(x) \leq 0 \\ &\qquad \underline{x} \leq x \leq \bar{x} \end{aligned}$$
Tracking Problem	The software formulates the tracking objective $F(x)$ as described in "Tracking Problem" on page 3-6 and minimizes the tracking objective:

Problem Type	Problem Formulation
	$$\min_{x} F(x)$$ $$s.t. \ \underline{x} \le x \le \bar{x}$$
Mixed Feasibility and Tracking Problem	The software minimizes following problem formulation: $$\min_{x} \ F(x)$$ $$s.t. \quad C(x) \le 0$$ $$\underline{x} \le x \le \bar{x}$$ **Note:** When tracking a reference signal, the software ignores the maximally feasible solution option.

Simplex Search Method Problem Formulations

The Simplex Search method uses the Optimization Toolbox function `fminsearch` and `fminbnd` to optimize model parameters to meet design requirements. `fminbnd` is used if one scalar parameter is being optimized, otherwise `fminsearch` is used. You cannot use parameter bounds $\underline{x} \le x \le \bar{x}$ with `fminsearch`.

Problem Type	Problem Formulation
Feasibility Problem	The software formulates the constraint $C(x)$ as described in "Feasibility Problem and Constraint Formulation" on page 3-3 and then minimizes the maximum constraint violation: $$\min_{x} \max(C(x))$$
Tracking Problem	The software formulates the tracking objective $F(x)$ as described in "Tracking Problem" on page 3-6 and then minimizes the tracking objective: $$\min_{x} F(x)$$
Mixed Feasibility and Tracking Problem	The software formulates the problem in two steps:

Problem Type	Problem Formulation
	1 Finds a feasible solution. $$\min_{x} \, \max(C(x))$$ **2** Minimizes the tracking objective. The software uses the results from step 1 as initial guesses and maintains feasibility by introducing a discontinuous barrier in the optimization objective. $$\min_{x} \, \Gamma(x)$$ where $$\Gamma(x) = \begin{cases} \infty & \text{if } \max(C(x)) > 0 \\ F(x) & \text{otherwise.} \end{cases}$$

Pattern Search Method Problem Formulations

The Pattern Search method uses the Global Optimization Toolbox function `patternsearch` to optimize model parameters to meet design requirements.

Problem Type	Problem Formulation
Feasibility Problem	The software formulates the constraint $C(x)$ as described in "Feasibility Problem and Constraint Formulation" on page 3-3 and then minimizes the maximum constraint violation: $$\min_{x} \, \max(C(x))$$ $$s.t. \quad \underline{x} \le x \le \bar{x}$$
Tracking Problem	The software formulates the tracking objective $F(x)$ as described in "Tracking Problem" on page 3-6 and then minimizes the tracking objective: $$\min_{x} \, F(x)$$ $$s.t. \quad \underline{x} \le x \le \bar{x}$$

Problem Type	Problem Formulation
Mixed Feasibility and Tracking Problem	The software formulates the problem in two steps: **1** Finds a feasible solution. $$\min_{x} \ \max(C(x))$$ $$s.t. \quad \underline{x} \le x \le \overline{x}$$ **2** Minimizes the tracking objective. The software uses the results from step 1 as initial guesses and maintains feasibility by introducing a discontinuous barrier in the optimization objective. $$\min_{x} \Gamma(x)$$ $$s.t. \ \underline{x} \le x \le \overline{x}$$ where $$\Gamma(x) = \begin{cases} \infty & \text{if } \max(C(x)) > 0 \\ F(x) & \text{otherwise.} \end{cases}$$

Gradient Computations

For the `Gradient descent` (`fmincon`) optimization solver, the gradients are computed using numerical perturbation:

$$dx = \sqrt[3]{eps} \times \max\left(|x|, \frac{1}{10} x_{typical} \right)$$

$$dL = \max(x - dx, x_{\min})$$

$$dR = \min(x + dx, x_{\max})$$

$$F_L = opt\_fcn(dL)$$

$$F_R = opt\_fcn(dR)$$

$$\frac{dF}{dx} = \frac{(F_L - F_R)}{(dL - dR)}$$

- x is a scalar design variable.

- x_{min} is the lower bound of x.
- x_{max} is the upper bound of x.
- $x_{typical}$ is the scaled value of x.
- opt_fcn is the objective function.

dx is relatively large to accommodate simulation solver tolerances.

If you want to compute the gradients in any other way, you can do so in the cost function you write for performing design optimization programmatically. See `sdo.optimize` and GradFcn of `sdo.OptimizeOptions` for more information.

Specify Signals to Log

Design requirements require logged model signals. During optimization, the model is simulated using the current value of the design variables and the logged signal is used to evaluate the design requirements.

1 **In the Response Optimization tool**, select **Signal** in the **New** drop-down list. A window opens where you select a signal to log.

2 In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

3 Select the signal and click to add it to the signal set.

4 In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

See Also

• "Design Optimization to Track Reference Signal (GUI)"

• sdo.SimulationTest

Specifying Step Response Characteristics

Specify Step Response Characteristics

To specify step response characteristics:

1   You can apply this requirement to either a signal or a linearization of your model.

   In the Response Optimization Tool, click **New**. To apply this requirement to a signal, select the **Step Response Envelope** entry in the **New Time Domain Requirement** section of the **New** list. To apply this requirement to a linearization of your model, select the **Step Response Envelope** entry in the **New Frequency Domain Requirement** section of the **New** list. The latter option requires Simulink Control Design software.

   A window opens where you specify the step response requirements on a signal, or system.

2   Specify a requirement name in the **Name** box.

3   Specify the step response characteristics:

- **Initial value**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final value**: Input level after the step occurs
- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Rise**: The percentage used in the rise time.
- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.
- **% Settling**: The percentage used in the settling time.
- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

4 Specify the signals or systems to be bound.

You can apply this requirement to a model signal or to a linearization of your Simulink model (requires Simulink Control Design software).

- Apply this requirement to a model signal:

  In the **Select Signals to Bound** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check-box.

  If you haven't selected a signal to log:

  a Click ⊞. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click [→] to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

- Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Alternatively, you can use the Check Step Response Characteristics block to specify step response bounds for a signal.

See Also

"Design Optimization to Meet Step Response Requirements (GUI)"

More About

· "Specify Time-Domain Design Requirements" on page 3-23
· "Specify Frequency-Domain Design Requirements" on page 3-39

Specifying Custom Requirements

To specify custom requirements, such as minimizing system energy:

1 In the Response Optimization tool, select **Custom Requirement** in the **New** list. A window opens where you specify the custom requirement.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the name of the function that contains the custom requirement in the **Function** box. The field must be specified as a function handle using @. The function must be on the MATLAB path. Click [folder icon] to review or edit the function.

If the function does not exist, clicking [folder icon] opens a template MATLAB file. Use this file to implement the custom requirement. The default function name is `myCustomRequirement`.

5 (Optional) If you want to prevent the solver from considering specific parameter combinations, select the **Error if constraint is violated** check box. Use this option for parameter-only constraints.

During an optimization iteration, the solver evaluates requirements with this option selected first.

- If the constraint is violated, the solver skips evaluating any remaining requirements and proceeds to the next iterate.

- If the constraint is *not* violated, the solver evaluates the remaining requirements for the current iterate. If any of the remaining requirements bound signals or systems, then the solver simulates the model .

For more information, see "Skip Model Simulation Based on Parameter Constraint Violation (GUI)".

Note: If you select this check box, then do not specify signals or systems to bound. If you *do* specify signals or systems, then this check box is ignored.

---

6 (Optional) Specify the signal or system, or both, to be bound.

You can apply this requirement to model signals, or a linearization of your Simulink model (requires Simulink Control Design software), or both.

Click **Select Signals and Systems to Bound (Optional)** to view the signal and linearization I/O selection area.

- Apply this requirement to a model signal:

  In the **Signal** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check box.

  If you have not selected a signal to log:

  a Click ⊞. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

  The window updates and displays the name of the block and the port number where the selected signal is located.

  c Select the signal and click ⇨ to add it to the signal set.

  d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

- Apply this requirement to a linear system.

  Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

  a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

  b Select the linearization input/output set from the **Linearization I/O** area.

  If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

  If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box. For more information on using this dialog box, see "Create Linearization I/O Sets".

  For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Related Examples

- "Design Optimization to Meet a Custom Objective (GUI)" on page 3-109
- "Design Optimization to Meet Custom Signal Requirements (GUI)" on page 3-136
- "Specify Time-Domain Design Requirements" on page 3-23
- "Specify Frequency-Domain Design Requirements" on page 3-39

Move Constraints

In this section...
"Move Constraints Graphically" on page 3-20
"Position Constraints Exactly" on page 3-21

Constraint-bound edges define time-domain constraints you would like to place on a particular signal in your model. You can position these edges, which appear as a yellow shaded region bordered by a black line, graphically or exactly.

Move Constraints Graphically

Use the mouse to click and drag edges in the amplitude versus time plot, as shown in the following figure.
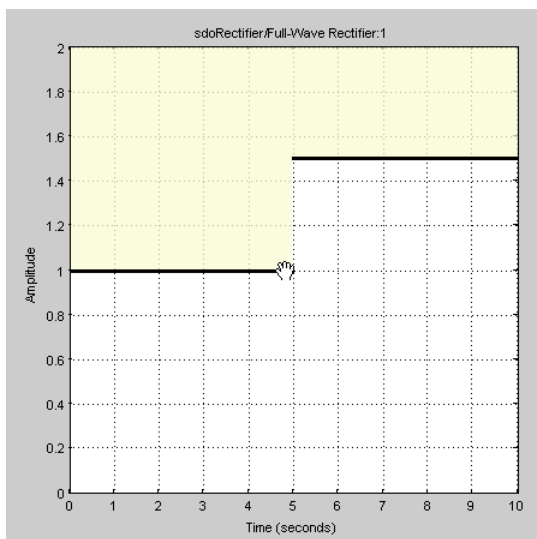
- To move a constraint edge boundary or to change the slope of a constraint edge, position the pointer over a constraint edge endpoint, and press and hold down the left mouse button. The pointer should change to a hand symbol. While still holding the button down, drag the pointer to the target location, and release the mouse button. Note that the edges on either side of the boundary might not maintain their slopes.
- To move an entire constraint edge up, down, left, or right, position the mouse pointer over the edge and press and hold down the left mouse button. The pointer should

change to a four-way arrow. While still holding the button down, drag the pointer to the target location, and release the mouse button. Note that the edges on either side of the boundary might not maintain their slopes.

To move a constraint edge to a perfectly horizontal or vertical position, hold down the **Shift** key while clicking and dragging the constraint edge. This causes the constraint edge to *snap* to a horizontal or vertical position.

When moving constraint bound edges, it is sometimes helpful to display gridlines on the axes for careful alignment of the constraint bound edges. To turn the gridlines on or off, right-click within the axes and select **Grid**.

Note: You can move a lower bound constraint edge above an upper bound constraint edge, or vice versa, but this produces an error when you attempt to run the optimization.

Position Constraints Exactly

To position a constraint edge exactly:

1 Position the pointer over the edge you want to move and right-click. Select **Edit** to open the Edit Design Requirement dialog box.
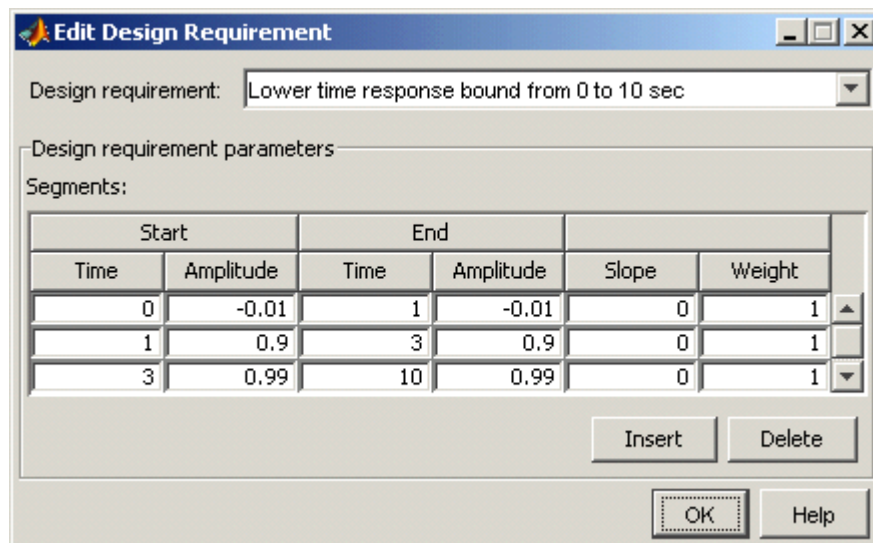
2   Specify the position of each constraint edge in the **Time** and **Amplitude** columns.

More About

- "Specify Time-Domain Design Requirements" on page 3-23
- "Specify Frequency-Domain Design Requirements" on page 3-39

Specify Time-Domain Design Requirements

In this section...

Specify Piecewise-Linear Lower and Upper Bounds

To specify upper and lower bounds on a signal:

1 In the Response Optimization tool, select **Signal Bound** in the **New** drop-down list. A window opens where you specify upper or lower bounds on a signal.

2 Specify a requirement name in the **Name** box.

3 Select the requirement type using the **Type** list.

4 Specify the edge start and end times and corresponding amplitude in the **Time (s)** and **Amplitude** columns.

5 Click [+] to specify additional bound edges.

   Select a row and click [🗑] to delete a bound edge.

6 In the **Select Signals to Bound** area, select a logged signal to apply the requirement to.

   If you have already selected signals, as described in "Specify Signals to Log" on page 3-12, they appear in the list. Select the corresponding check-box.

   If you haven't selected a signal to log:

   a Click [+]. A window opens where you specify the logged signal.

   b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

7 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can add a Check Custom Bounds block to your model to specify piecewise-linear bounds.

Specify Signal Property Requirements

To specify signal property requirements:

1. In the Response Optimization tool, select **Signal Property** in the **New** drop-down list. The Create Requirement window opens where you specify signal property requirements.

2. In the **Name** box, specify a requirement name.

3. In the **Specify Property** area, specify a signal property requirement using the **Property** and **Type** lists and the **Bound** box.

Property List

For a signal $S(t_0), \ldots, S(t_N)$ you can specify one of the following properties using the **Property** list:

- `Signal minimum` — $min(S)$
- `Signal maximum` — $max(S)$
- `Signal final value` — $S(t_N)$
- `Signal mean` — $mean(S)$
- `Signal median` — $median(S)$
- `Signal variance` — $variance(S)$
- `Signal interquartile range` — Difference between the 75th and 25th percentiles of the signal values.
- `Signal sum` — $\displaystyle\sum_{i=t_0}^{t_N} S(i)$
- `Signal sum square` — $\displaystyle\sum_{i=t_0}^{t_N} S(i)^2$
- `Signal sum absolute` — $\displaystyle\sum_{i=t_0}^{t_N} |S(i)|$

Custom Signal Property

You can add a custom signal property to the **Property** list by editing the function `sdo.requirements.signalPropertyFcns`.

a At the MATLAB command prompt, enter `edit sdo.requirements.signalPropertyFcns`.

b Add your signal property function to the `FcnData` cell array.

Your signal property function must be on the path.

4 In the **Select Signals to Bound** area, select the logged signal to which you want to apply the requirement.

If you have already selected a signal, as described in "Specify Signals to Log" on page 3-12, the signal appears in the list. Select the corresponding check box for that signal.

If you have not selected a signal to log:

a Click [+]. The Create Signal Set window opens where you specify the logged signal.

b In the Simulink model window, click the signal to which you want to add a requirement.
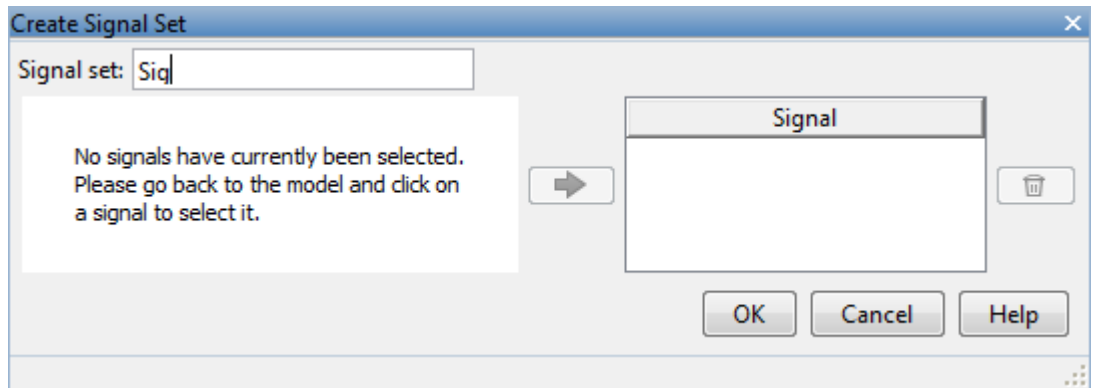
The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click [⇨] to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

5   Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. An iteration plot depicting the signal property for each iteration also appears in the Response Optimization tool window.

Specify Step Response Characteristics

To specify step response characteristics:

1   You can apply this requirement to either a signal or a linearization of your model.

In the Response Optimization Tool, click **New**. To apply this requirement to a signal, select the **Step Response Envelope** entry in the **New Time Domain Requirement** section of the **New** list. To apply this requirement to a linearization of your model, select the **Step Response Envelope** entry in the **New Frequency Domain Requirement** section of the **New** list. The latter option requires Simulink Control Design software.

A window opens where you specify the step response requirements on a signal, or system.

2   Specify a requirement name in the **Name** box.

3   Specify the step response characteristics:

- **Initial value**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final value**: Input level after the step occurs
- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Rise**: The percentage used in the rise time.
- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.
- **% Settling**: The percentage used in the settling time.
- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

4 Specify the signals or systems to be bound.

You can apply this requirement to a model signal or to a linearization of your Simulink model (requires Simulink Control Design software).

- Apply this requirement to a model signal:

  In the **Select Signals to Bound** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check-box.

  If you haven't selected a signal to log:

  a Click ⌞ + ⌟. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

  The window updates and displays the name of the block and the port number where the selected signal is located.

  c Select the signal and click ⌞ ⇨ ⌟ to add it to the signal set.

  d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

- Apply this requirement to a linear system.

  Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

  a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

  b Select the linearization input/output set from the **Linearization I/O** area.

  If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

  If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box.

  For more information on using this dialog box, see "Create Linearization I/O Sets".

  For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Alternatively, you can use the Check Step Response Characteristics block to specify step response bounds for a signal.

See Also

"Design Optimization to Meet Step Response Requirements (GUI)"

Track Reference Signals

Use reference tracking to force a model signal to match a desired signal.

To track a reference signal:

1 In the Response Optimization tool, select **Signal Tracking** in the **New** drop-down list. A window opens where you specify the reference signal to track.

2 Specify a requirement name in the **Name** box.

3 Define the reference signal by entering vectors, or variables from the workspace, in the **Time vector** and **Amplitude** fields.

   Click **Update reference signal data** to use the new amplitude and time vector as the reference signal.

4 Specify how the optimization solver minimizes the error between the reference and model signals using the **Tracking Method** list:

   • SSE — Reduces the sum of squared errors

   • SAE — Reduces the sum of absolute errors

5 In the **Specify Signal to Track Reference Signal** area, select a logged signal to apply the requirement to.

   If you already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, they appear in the list. Select the corresponding check-box.

   If you haven't selected a signal to log:

   a Click [+]. A window opens where you specify the logged signal.

   b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c
Select the signal and click [→] to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

e Select the check-box corresponding to the signal and click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the signal bound also appears in the Response Optimization tool window.

Note: When tracking a reference signal, the software ignores the maximally feasible solution option. For more information on this option, see "Selecting Optimization Termination Options" on page 3-75.

Alternatively, you can use the Check Against Reference block to specify a reference signal to track.

See Also

"Design Optimization to Track Reference Signal (GUI)"

Specify Custom Requirements

To specify custom requirements, such as minimizing system energy:

1 In the Response Optimization tool, select **Custom Requirement** in the **New** list. A window opens where you specify the custom requirement.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the name of the function that contains the custom requirement in the **Function** box. The field must be specified as a function handle using @. The function must be on the MATLAB path. Click [📁] to review or edit the function.

If the function does not exist, clicking [📁] opens a template MATLAB file. Use this file to implement the custom requirement. The default function name is `myCustomRequirement`.

5 (Optional) If you want to prevent the solver from considering specific parameter combinations, select the **Error if constraint is violated** check box. Use this option for parameter-only constraints.

During an optimization iteration, the solver evaluates requirements with this option selected first.

- If the constraint is violated, the solver skips evaluating any remaining requirements and proceeds to the next iterate.

- If the constraint is *not* violated, the solver evaluates the remaining requirements for the current iterate. If any of the remaining requirements bound signals or systems, then the solver simulates the model .

For more information, see "Skip Model Simulation Based on Parameter Constraint Violation (GUI)".

Note: If you select this check box, then do not specify signals or systems to bound. If you *do* specify signals or systems, then this check box is ignored.

---

6 (Optional) Specify the signal or system, or both, to be bound.

You can apply this requirement to model signals, or a linearization of your Simulink model (requires Simulink Control Design software), or both.

Click **Select Signals and Systems to Bound (Optional)** to view the signal and linearization I/O selection area.

- Apply this requirement to a model signal:

  In the **Signal** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check box.

  If you have not selected a signal to log:

a Click [+]. A window opens where you specify the logged signal.

b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click [→] to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

• Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box. For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

See Also

- "Design Optimization to Meet a Custom Objective (GUI)" on page 3-109
- "Design Optimization to Meet Custom Signal Requirements (GUI)" on page 3-136

Edit Design Requirements

The Edit Design Requirement dialog box allows you to exactly position constraint segments and to edit other properties of these constraints. The dialog box has two main components:

- An upper panel to specify the constraint you are editing
- A lower panel to edit the constraint parameters

The upper panel of the Edit Design Requirement dialog box resembles the image in the following figure.

Design requirement: Upper time response bound from 0 to 10 sec ▼

In the context of the SISO Tool in Control System Toolbox™ software, **Design requirement** refers to both the particular editor within the SISO Tool that contains the requirement and the particular requirement within that editor. To edit other constraints within the SISO Tool, select another design requirement from the drop-down menu.

Edit Design Requirement Dialog Box Parameters

The particular parameters shown within the lower panel of the Edit Design Requirement dialog box depend on the type of constraint/requirement. In some cases, the lower panel

contains a grid with one row for each segment and one column for each constraint parameter. The following table summarizes the various constraint parameters.

Edit Design Requirement Dialog Box Parameters

Parameter	Found in	Description
Time	Upper and lower time response bounds on step and impulse response plots	Defines the time range of a segment within a constraint/requirement.
Amplitude	Upper and lower time response bounds on step and impulse response plots	Defines the beginning and ending amplitude of a constraint segment.
Slope (1/s)	Upper and lower time response bounds	Defines the slope, in 1/s, of a constraint segment. It is an alternative method of specifying the magnitude values. Entering a new **Slope** value changes any previously defined magnitude values.
Final value	Step response bounds	Defines the input level after the step occurs.
Rise time	Step response bounds	Defines a constraint segment for a particular rise time.
% Rise	Step response bounds	The percentage of the step's range used to describe the rise time.
Settling time	Step response bounds	Defines a constraint segment for a particular settling time.
% Settling	Step response bounds	The percentage of the final value that defines the settling region used to describe the settling time.
% Overshoot	Step response bounds	
% Undershoot	Step response bounds	Defines the constraint segments for a particular percent undershoot.

Edit Design Requirements

The Edit Design Requirement dialog box allows you to exactly position constraint segments and to edit other properties of these constraints. The dialog box has two main components:

- An upper panel to specify the constraint you are editing
- A lower panel to edit the constraint parameters

The upper panel of the Edit Design Requirement dialog box resembles the image in the following figure.

Design requirement: Upper time response bound from 0 to 10 sec ▼

In the context of the SISO Tool in Control System Toolbox software, **Design requirement** refers to both the particular editor within the SISO Tool that contains the requirement and the particular requirement within that editor. To edit other constraints within the SISO Tool, select another design requirement from the drop-down menu.

Edit Design Requirement Dialog Box Parameters

The particular parameters shown within the lower panel of the Edit Design Requirement dialog box depend on the type of constraint/requirement. In some cases, the lower panel contains a grid with one row for each segment and one column for each constraint parameter. The following table summarizes the various constraint parameters.

Edit Design Requirement Dialog Box Parameters

Parameter	Found in	Description
Time	Upper and lower time response bounds on step and impulse response plots	Defines the time range of a segment within a constraint/requirement.
Amplitude	Upper and lower time response bounds on step and impulse response plots	Defines the beginning and ending amplitude of a constraint segment.
Slope (1/s)	Upper and lower time response bounds	Defines the slope, in 1/s, of a constraint segment. It is an alternative method

Parameter	Found in	Description
		of specifying the magnitude values. Entering a new **Slope** value changes any previously defined magnitude values.
Final value	Step response bounds	Defines the input level after the step occurs.
Rise time	Step response bounds	Defines a constraint segment for a particular rise time.
% Rise	Step response bounds	The percentage of the step's range used to describe the rise time.
Settling time	Step response bounds	Defines a constraint segment for a particular settling time.
% Settling	Step response bounds	The percentage of the final value that defines the settling region used to describe the settling time.
% Overshoot	Step response bounds	
% Undershoot	Step response bounds	Defines the constraint segments for a particular percent undershoot.

Specify Frequency-Domain Design Requirements

Specify Lower Bounds on Gain and Phase Margin

To specify lower bounds on the gain and phase margin of a linear system:

1 In the Response Optimization tool, select **Gain and Phase Margin** in the **New** list. A window opens where you specify lower bounds on the gain and phase margin of your linear system.

2 Specify a requirement name in **Name**.

3 Specify bounds on the gain margin or phase margin, or both.

Bode Diagram
Gm = 9.54 dB (at 2.24 rad/s) , Pm = 25.4 deg (at 1.23 rad/s)

- **Gain margin** — Amount of gain increase or decrease required to make the loop gain unity at the frequency where the phase angle is –180°.

- **Phase margin** — Amount of phase increase or decrease required to make the phase angle –180° when the loop gain is 1.0

To specify a lower bound on the gain margin or phase margin, or both, select the corresponding check box and enter the lower bound value.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5   Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6   (Optional) In the graphical display, you can:

   · "Move Constraints Graphically"
   · "Position Constraints Exactly"

Alternatively, you can use the Check Gain and Phase Margins block to specify bounds on the gain and phase margin. (Requires Simulink Control Design.)

Specify Piecewise-Linear Lower and Upper Bounds on Frequency Response

To specify upper or lower bounds on the magnitude of a system response:

1   In the Response Optimization tool, select **Bode Magnitude** in the **New** list. A window opens where you specify the lower or upper bounds on the magnitude of the system response.

2   Specify a requirement name in the **Name** box.

3   Specify the requirement type using the **Type** list.

4   Specify the edge start and end frequencies and corresponding magnitude in the **Frequency** and **Magnitude** columns.

5   Insert or delete bound edges.

Click [+] to specify additional bound edges.

Select a row and click [🗑] to delete a bound edge.

6 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

    a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

    b Select the linearization input/output set from the **Linearization I/O** area.

       If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

       If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

       For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool window. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Bode Characteristics block to specify bounds on the magnitude of the system response. (Requires Simulink Control Design.)

Specify Bound on Closed-Loop Peak Gain

To specify an upper bound on the closed-loop peak response of a system:

1 In the Response Optimization tool, select **Closed-Loop Peak Gain** in the **New** list. A window opens where you specify an upper bound on the closed-loop peak gain of the system.

2 Specify a requirement name in the **Name** box.

3 Specify the upper bound on the closed-loop peak gain in the **Closed-Loop peak gain** box.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click ⊞ to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool window. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Nichols Characteristics block to specify bounds on the magnitude of the system response. (Requires Simulink Control Design.)

Specify Lower Bound on Damping Ratio

To specify a lower bound on the damping ratio of the system:

1 In the Response Optimization tool, select **Damping Ratio** in the **New** list. A window opens where you specify an upper bound on the damping ratio of the system.

2 Specify a requirement name in the **Name** box.

3 Specify the lower bound on the damping ratio in the **Damping ratio** box.

4   In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b   Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5   Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.
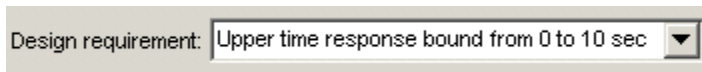
pidtune_demo/Sum:1 [in], pidtune_demo/Plant:1 [out,Open Loop]

6   (Optional) In the graphical display, you can:

   • "Move Constraints Graphically"

   • "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify a bound on the damping ratio. (Requires Simulink Control Design.)

Specify Upper and Lower Bounds on Natural Frequency

To specify a bound on the natural frequency of the system:

1   In the Response Optimization tool, select **Natural Frequency** in the **New** list. A window opens where you specify a bound on the natural frequency of the system.

2   Specify a requirement name in the **Name** box.

3   Specify a lower or upper bound on the natural frequency in the **Natural frequency** box.

4   In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b   Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5   Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify a bound on the natural frequency. (Requires Simulink Control Design.)

Specify Upper Bound on Approximate Settling Time

To specify an upper bound on the approximate settling time of the system:

1 In the Response Optimization tool, select **Settling Time** in the **New** list. A window opens where you specify an upper bound on the approximate settling time of the system.

2 Specify a requirement name in the **Name** box.

3   Specify the upper bound on the approximate settling time in the **Settling time** box.

4   In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

      Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

      a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

      b   Select the linearization input/output set from the **Linearization I/O** area.

          If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

          If you have not created a linearization input/output set, click $\boxed{+}$ to open the Create linearization I/O set dialog box.

          For more information on using this dialog box, see "Create Linearization I/O Sets".

      For more information on linearization, see "What Is Linearization?".

5   Click **OK**.

      A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify the approximate settling time. (Requires Simulink Control Design.)

Specify Piecewise-Linear Upper and Lower Bounds on Singular Values

To specify piecewise-linear upper and lower bounds on the singular values of a system:

1 In the Response Optimization tool, select **Singular Values** in the **New** list. A window opens where you specify the lower or upper bounds on the singular values of the system.

2 Specify a requirement name in the **Name** box.

3   Specify the requirement type using the **Type** list.

4   Specify the edge start and end frequencies and corresponding magnitude in the **Frequency** and **Magnitude** columns, respectively.

5   Insert or delete bound edges.

     Click ⊞ to specify additional bound edges.

     Select a row and click 🗑 to delete a bound edge.

6   In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

     Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

     a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

     b   Select the linearization input/output set from the **Linearization I/O** area.

        If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

        If you have not created a linearization input/output set, click ⊞ to open the Create linearization I/O set dialog box.

        For more information on using this dialog box, see "Create Linearization I/O Sets".

     For more information on linearization, see "What Is Linearization?".

7   Click **OK**.

     A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Singular Value Characteristics block to specify bounds on the singular value. (Requires Simulink Control Design.)

Specify Step Response Characteristics

To specify step response characteristics:

1 You can apply this requirement to either a signal or a linearization of your model.

In the Response Optimization Tool, click **New**. To apply this requirement to a signal, select the **Step Response Envelope** entry in the **New Time Domain**

Requirement section of the **New** list. To apply this requirement to a linearization of your model, select the **Step Response Envelope** entry in the **New Frequency Domain Requirement** section of the **New** list. The latter option requires Simulink Control Design software.

A window opens where you specify the step response requirements on a signal, or system.

2 Specify a requirement name in the **Name** box.

3 Specify the step response characteristics:

- **Initial value**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final value**: Input level after the step occurs
- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Rise**: The percentage used in the rise time.

- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.

- **% Settling**: The percentage used in the settling time.

- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

4 Specify the signals or systems to be bound.

You can apply this requirement to a model signal or to a linearization of your Simulink model (requires Simulink Control Design software).

- Apply this requirement to a model signal:

  In the **Select Signals to Bound** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check-box.

  If you haven't selected a signal to log:

  a Click [+]. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c   Select the signal and click ⬜ to add it to the signal set.

d   In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

•   Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b   Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click ⬜ to open the Create linearization I/O set dialog box.
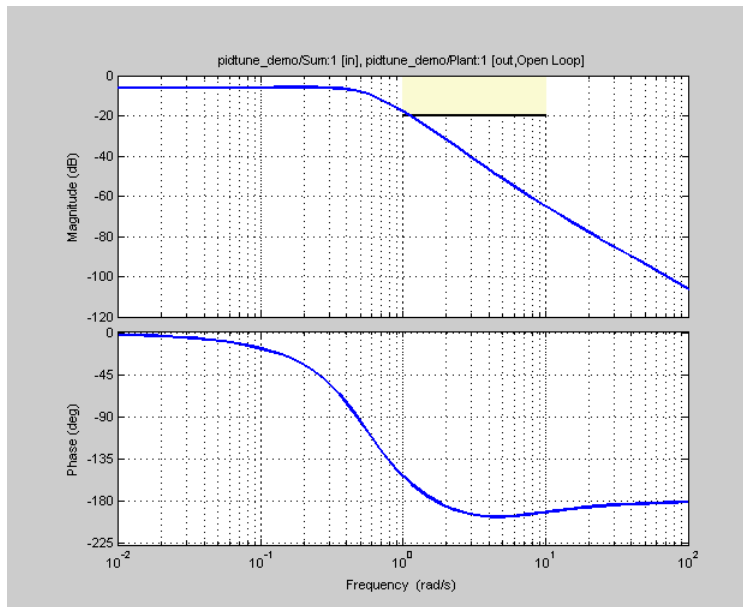
For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Alternatively, you can use the Check Step Response Characteristics block to specify step response bounds for a signal.

See Also

"Design Optimization to Meet Step Response Requirements (GUI)"

Specify Custom Requirements

To specify custom requirements, such as minimizing system energy:

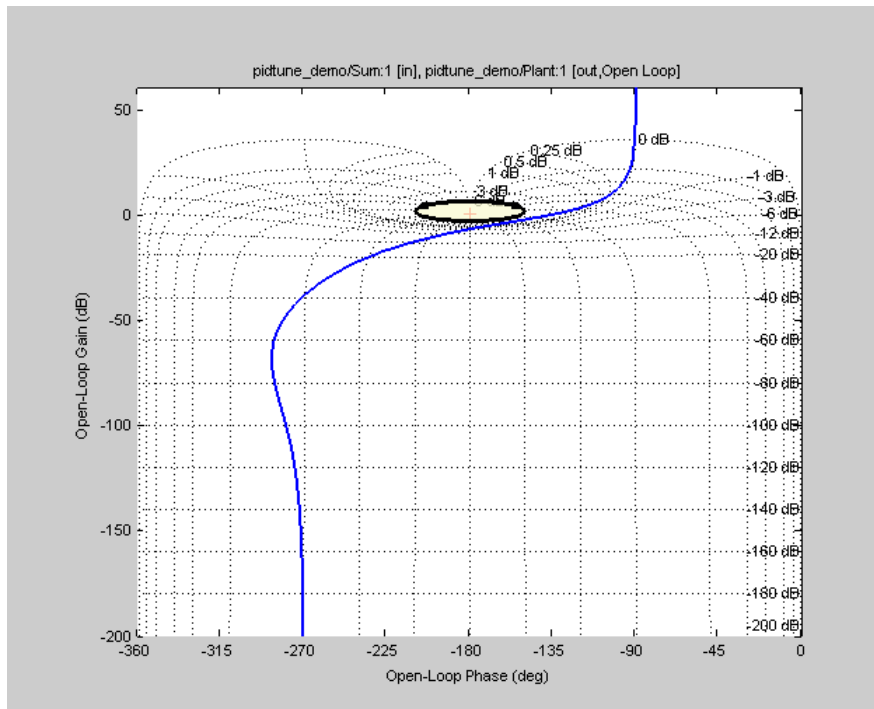**1** In the Response Optimization tool, select **Custom Requirement** in the **New** list. A window opens where you specify the custom requirement.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the name of the function that contains the custom requirement in the **Function** box. The field must be specified as a function handle using @. The function

must be on the MATLAB path. Click [folder icon] to review or edit the function.

If the function does not exist, clicking [folder icon] opens a template MATLAB file. Use this file to implement the custom requirement. The default function name is `myCustomRequirement`.

5 (Optional) If you want to prevent the solver from considering specific parameter combinations, select the **Error if constraint is violated** check box. Use this option for parameter-only constraints.

During an optimization iteration, the solver evaluates requirements with this option selected first.

- If the constraint is violated, the solver skips evaluating any remaining requirements and proceeds to the next iterate.

- If the constraint is *not* violated, the solver evaluates the remaining requirements for the current iterate. If any of the remaining requirements bound signals or systems, then the solver simulates the model .

For more information, see "Skip Model Simulation Based on Parameter Constraint Violation (GUI)".

Note: If you select this check box, then do not specify signals or systems to bound. If you *do* specify signals or systems, then this check box is ignored.

---

6 (Optional) Specify the signal or system, or both, to be bound.

You can apply this requirement to model signals, or a linearization of your Simulink model (requires Simulink Control Design software), or both.

Click **Select Signals and Systems to Bound (Optional)** to view the signal and linearization I/O selection area.

- Apply this requirement to a model signal:

  In the **Signal** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check box.

  If you have not selected a signal to log:

  a Click ⊞. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

· Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box. For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

See Also

- "Design Optimization to Meet a Custom Objective (GUI)" on page 3-109
- "Design Optimization to Meet Custom Signal Requirements (GUI)" on page 3-136

Specify Design Variables

Before running the optimization, you must define the model parameters to optimize. These parameters form the *design variables set* for optimization. By tuning these parameters, Simulink Design Optimization software attempts to make the signals meet the requirements.

Simulink Design Optimization software optimizes the response signals of the model by varying the tuned parameters so that the response signals lie within the constraint bound segments or closely match a specified reference signal. The design variables can be scalar, vector, matrix, or an expression that evaluates to one of these values.

To specify the parameters to be tuned using the Response Optimization tool:

1   In the **Design Variables Set** list, select **New**.

    A window opens where you specify design variables. All model parameters are displayed in this window.

2   Select one or more parameter names and click

    to add the selected parameters to a design variables set.

    Note: You can add the same parameter to multiple design variable sets.

3   (Optional) Specify design variable settings.

Setting	Description	Default
Variable	The name of the parameter.	Not an editable field
Value	Value of the model parameter. This value is used by the optimization method as the initial value and is modified during optimization.	Current value of the parameter in the model. If you edit this column, click **Update model variable values** to update the values in the model.
Minimum	The minimum value or lower bound for the parameter. You can edit this field to provide an alternate minimum value.	`-Inf`

Setting	Description	Default
Maximum	The maximum value or upper bound for the parameter. You can edit this field to provide an alternate maximum value.	`Inf`
Scale	During optimization, the design variables are scaled, or normalized, by dividing their current value by a scale value. You can edit this field to provide an alternate scaling factor.	Next power of 2 greater than the current value of the parameter

The check-box indicates whether the parameter is selected as an design variable in the set. Select it if you want this parameter to be tuned during the optimization. Deselect if you do not want this parameter to be tuned during the optimization but you would like to keep it on the list of tuned parameters (for a subsequent optimization).

Expand **Variable Detail** to see the block in the model that contains this parameter.

4 Click **OK** to create a design variable set.

Related Examples

- "Specify Independent Parameters to Optimize" on page 3-63
- "Optimize Parameters for Robustness (GUI)" on page 3-206
- "Update Model with Design Variables Set" on page 3-67
- "Export Design Variable Values for Specific Iteration" on page 3-92

Specify Independent Parameters to Optimize

This example shows how to specify independent parameters, that do not appear explicitly in the model, as optimization parameters.

Assume that the parameter Kint in the model srotut1 is related to the parameters x and y according to the relationship Kint=x+y. Also assume that the initial values of x and y are 1 and -0.7, respectively. To tune x and y instead of Kint, first define these parameters in the model workspace. To do this,

1   Open the Simulink model.

    open_system('srotut1')

2   Add the independent parameters to the model workspace, along with their initial values.

    •   Select **View** > **Model Explorer** from the srotut1 window to open the Model Explorer window.

    •   In the Model Hierarchy tree, select **srotut1** > **Model Workspace**.
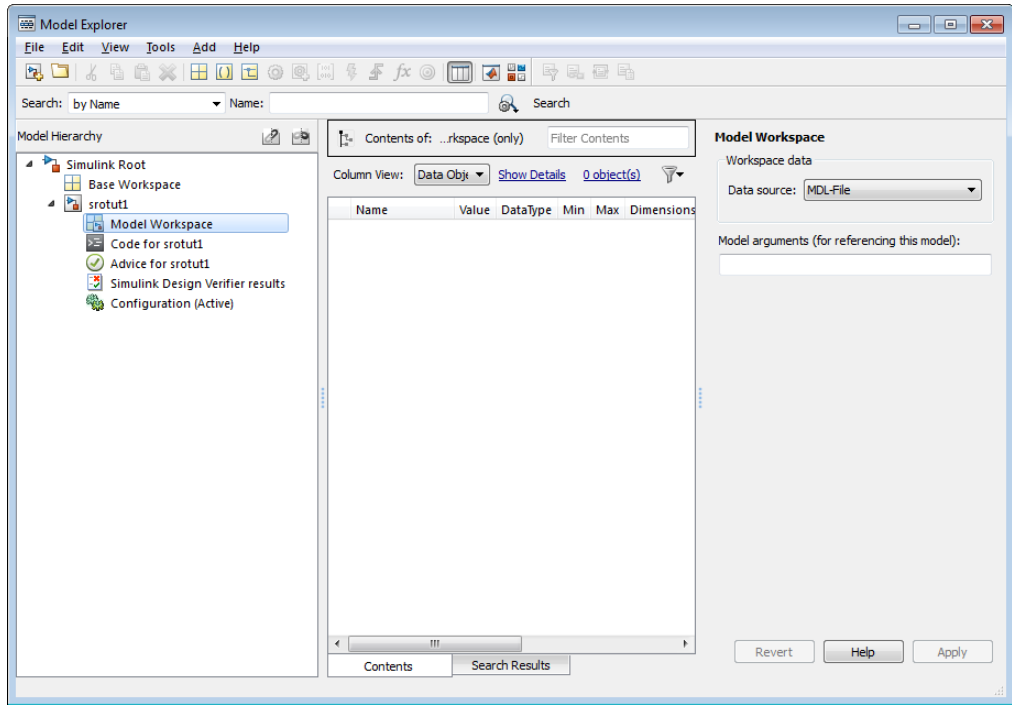
- Select **Add** > **MATLAB Variable** to add a new variable to the model workspace. A new variable with a default name `Var` appears in the **Name** column.

- Double-click `Var` to make it editable and change the variable name to `x`. Edit the initial **Value** to `1`.

- Repeat steps 3 and 4 to add a variable `y` with an initial value of `-0.7`.

  The Model Explorer window resembles the following figure.
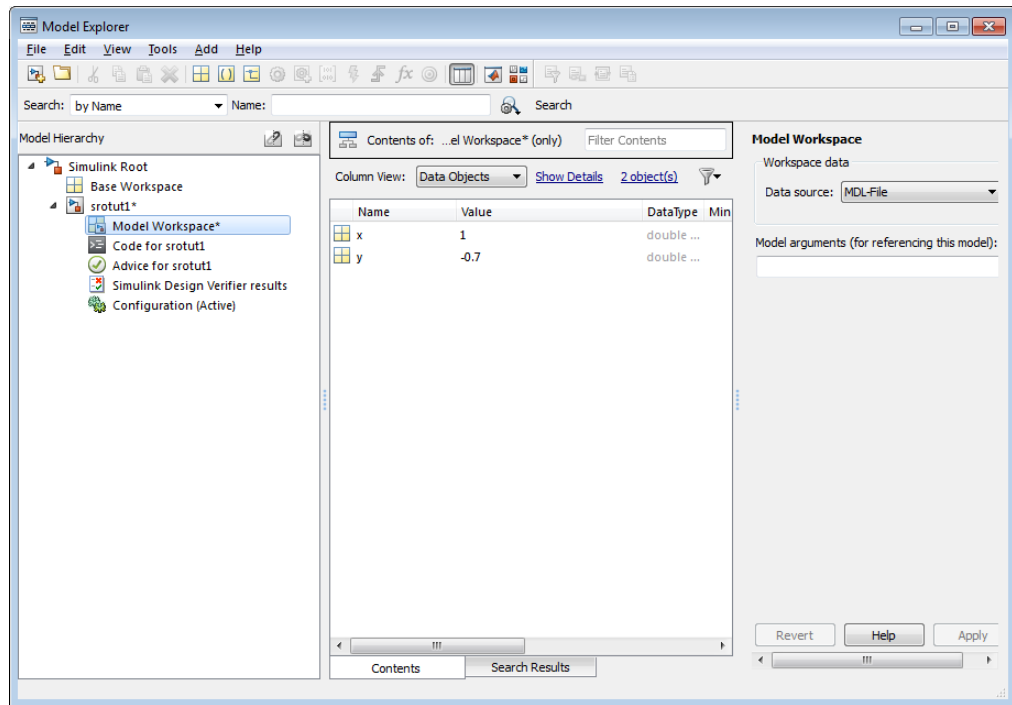
3   Define a Simulation Start function that runs before each simulation of the model. This Simulation Start function defines the relationship between the dependent parameters in the model and the independent parameters in the model workspace.

- In the `srotut1` model window, select **File** > **Model Properties** > **Model Properties**.

- In the Model Properties window, click the **Callbacks** tab.

- To enter a Simulation start function, select **StartFcn**, and type the name of a new function. For example, type `srotut1_start` in the **Simulation start function** panel. Then, click **OK**.

- Create a MATLAB file named `srotut1_start`.

  The content of the file defines the relationship between the parameters in the model and the parameters in the workspace. For this example, the content resembles the following:

```
wks = get_param(gcs, 'ModelWorkspace')
x = wks.evalin('x')
y = wks.evalin('y')
Kint = x+y;
```

Note: You must first use `get_param` to get the variables independent variables, x and y, from the model workspace. Then define the dependent variable, Kint, in terms of the independent variables.

When you add a new design or uncertain parameter, x and y appear in the dialog box.

Caution Avoid adding independent parameters together with their corresponding dependent parameters to the lists of tuned and uncertain parameters. Otherwise, the optimization could give incorrect results. For example, when a parameter c depends on the parameters a and b avoid adding all three parameters to the lists.

Related Examples

- "Specify Design Variables" on page 3-61

Update Model with Design Variables Set

This example shows how to update a model with a set of design variables.

Open the Simulink model and load the pre-configured Response Optimization tool session.

```
load('pidtune_demo_sdosession_update_dv.mat')
sdotool(SDOSessionData)
```

The Response Optimization tool opens and loads the preconfigured session. In the **Data** area, DesignVars1 is a set of tuned design variables.

In the **Design Variables Set** list, select the design variable set, DesignVars1.
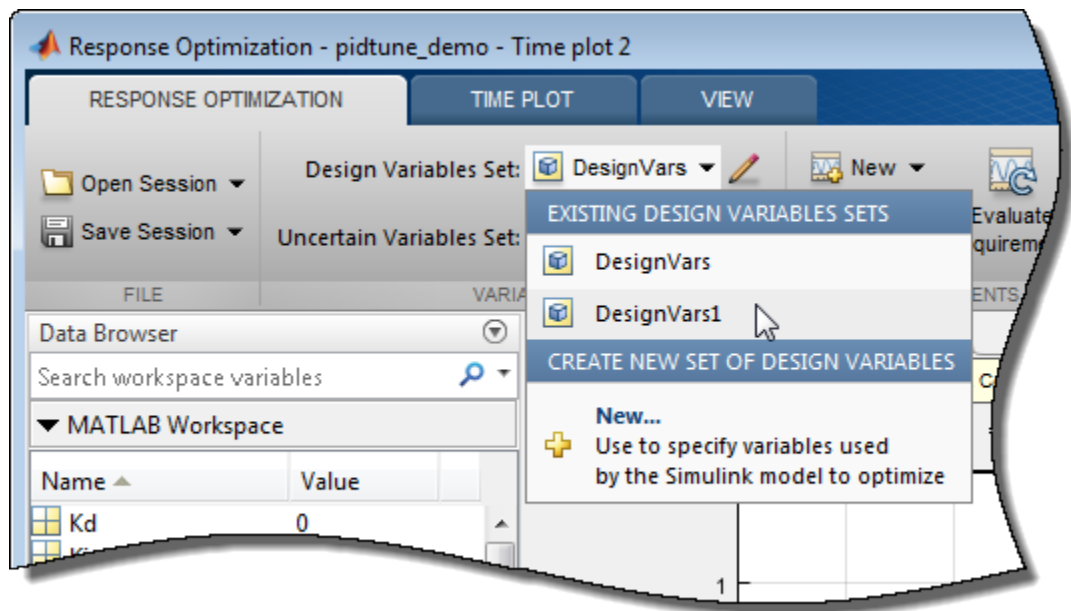
Open the Edit dialog box.

Click ![pencil icon] for the **Design Variable Set** list.

Select the variables you want to update in the model.

For this example, select Kd, Ki, and Kp.

	Variable	Value	Minimum	Maximum	Scale
☑	Kd	0.3624295...	-Inf	Inf	0
☑	Ki	0.1170876...	-Inf	Inf	0
☑	Kp	0.2855104...	-Inf	Inf	0.1

Edit: DesignVars1

Update model variables

▶ Variable Detail
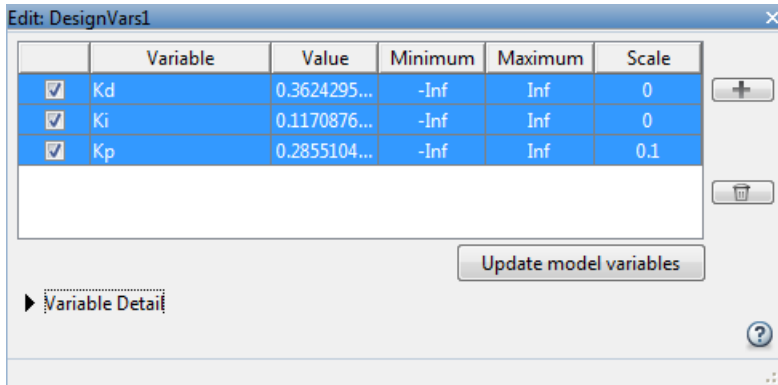
Click **Update model variables**.

Plot the model response.

In the **Response Optimization** tab, click **Plot Current Response**.

Related Examples

General Options

In this section...
"Accessing General Options" on page 3-69
"Progress Options" on page 3-69
"Result Options" on page 3-70

Accessing General Options

You can set optimization progress and result options. To set these options, click **Options** in the Response Optimization tool. A window opens. Select the **General Options** tab.

Response Optimization Options   ✕

| General Options | Optimization Options | Parallel Options | Linearization Options |

Progress Options

☑ Show optimization progress window during optimization

☑ Update plots during optimization

Result Options

☑ Update model at end of optimization

☐ Save optimization termination information

☑ Save requirement values at the end of optimization.

Design variable sets:

⦿ Overwrite optimized variable values in design variable set

◯ Save optimized variable values as new design variable set

OK   Cancel   Help

Progress Options

You can specify options related to optimization progress using the options in the **Progress Options** area.

Progress Options

☑ Show optimization progress window during optimization

☑ Update plots during optimization
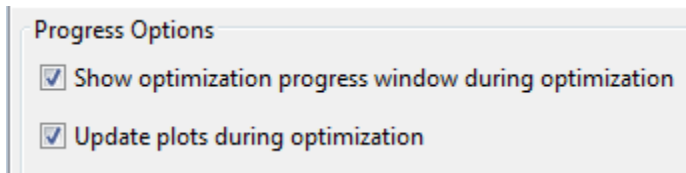
- **Show optimization progress window during optimization**

  Opens an optimization progress window during optimization. The window displays information such as constraint violations and cost function if **Display level** is Iteration. The window is updated at the end of the optimization with the termination information such as whether the optimization converged.

- **Update plots during optimization**

  Updates model response and design variable plots at each optimization iteration.

Result Options

You can specify options for optimization results in the **Result Options** area.

Result Options

☑ Update model at end of optimization

☐ Save optimization termination information

☑ Save requirement values at the end of optimization.

Design variable sets:

◉ Overwrite optimized variable values in design variable set

◯ Save optimized variable values as new design variable set
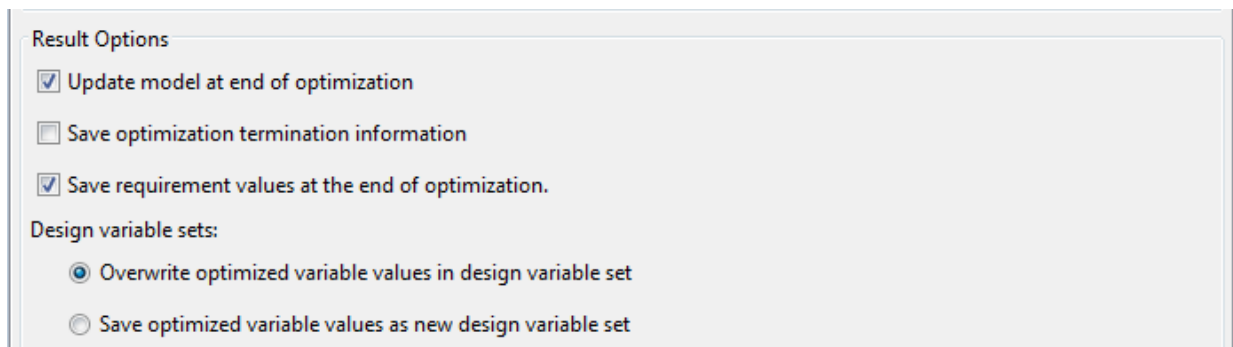
- **Update model at end of optimization**

  Updates optimized parameter values in the Simulink model after the optimization terminates.

- **Save optimization termination information**

  Saves termination information returned by the optimization solver as a variable named info in the **Data** area. info is a structure with one or more of the following fields:

- **F** — Optimized cost (objective) value.
- **Cleq** — Optimized nonlinear inequality constraint violations.

  The field appears if the optimization problem includes a nonlinear inequality constraint.

  The value is a `mx1` vector. Positive values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- **Ceq** — Optimized nonlinear equality constraint violations.

  The field appears if the optimization problem includes a nonlinear equality constraint.

  The value is a double `rx1` vector. Any nonzero values indicate that the constraint has not been satisfied. Check `exitflag` to confirm that the optimization succeeded.

- **Gradients** — Cost and constraint gradients at the optimized parameter values. See "How the Optimization Algorithm Formulates Minimization Problems" on how the solver computes gradients.

  This field appears if the solver specified in the `Method` property of `sdo.OptimizeOptions` computes gradients.

  The value is a structure.

- **exitflag** — Integer identifying the reason the algorithm terminated. See `fmincon`, `patternsearch` and `fminsearch` for a list of the values and the corresponding termination reasons.

- **iterations** — Number of optimization iterations

- **SolverOutput** — A structure with solver-specific output information. The fields of this structure depends on the optimization solver specified in the `Method` property of `sdo.OptimizeOptions`. See `fmincon`, `patternsearch` and `fminsearch` for a list of solver outputs and their description.

- **Stats** — A structure that contains statistics collected during optimization, such as start and end times, number of function evaluations and restarts.

- **Save requirement values at the end of optimization**

  Saves requirement values after the optimization terminates.

- **Design variable sets**

  - **Overwrite optimized variable values in design variable set**

    Overwrites the optimized model parameter values in the design variable set variable used in the optimization. You can see the updated values in the `Value` field.
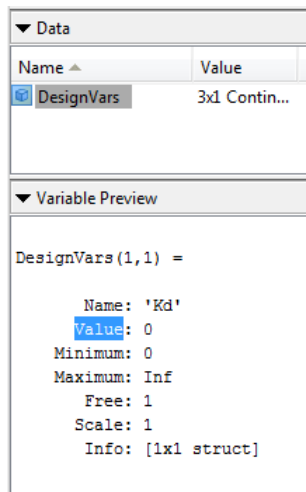
  - **Save optimized variable values as new design variable set**

    Creates a new variable in the **Data** area of the tool that contains the optimized parameter values. To update the model with the optimized parameter values, select the variable in the **Design Variables Set** list. Open the Edit dialog box by clicking 🖉. Select the parameters of interest and click **Update model variables**.

Optimization Options

In this section...
"Accessing Optimization Options" on page 3-73
"Selecting Optimization Methods" on page 3-74
"Selecting Optimization Termination Options" on page 3-75
"Selecting Additional Optimization Options" on page 3-76

Accessing Optimization Options

You can set several options for the optimization. These options include the optimization methods and the tolerances the methods use.

To set optimization options, click **Options** in the Response Optimization tool. A window opens. Select the **Optimization Options** tab.
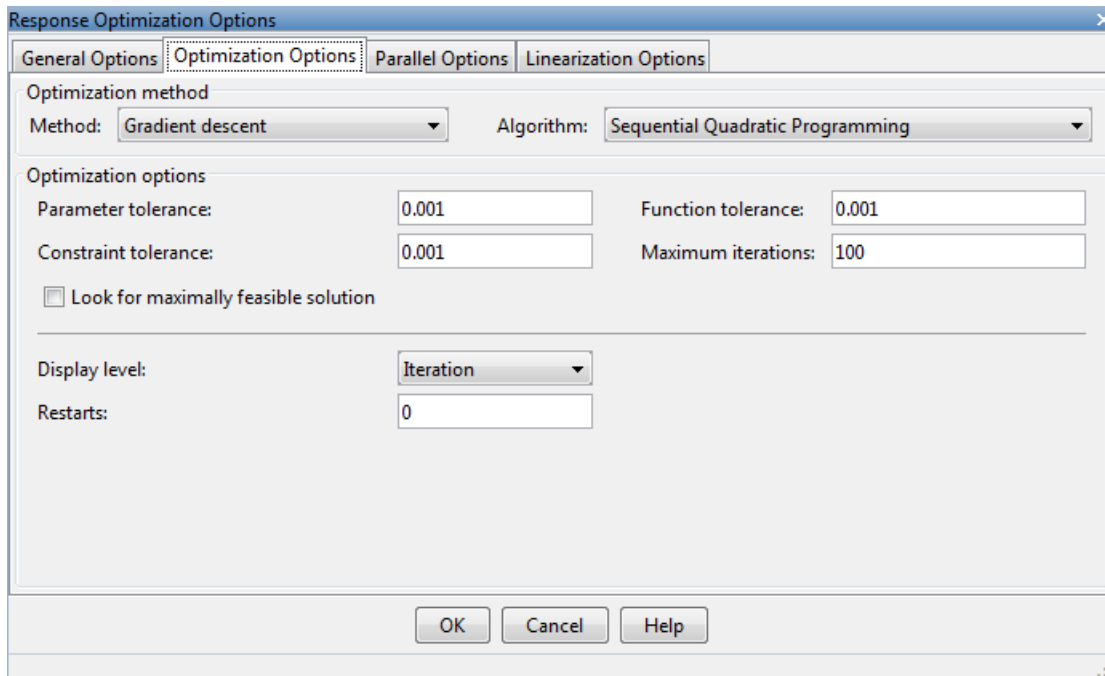
Selecting Optimization Methods

Both the **Method** and **Algorithm** options in the **Optimization method** area define the optimization method.
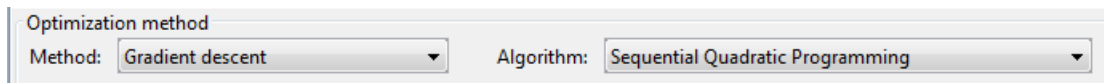
The choices for the **Method** option are:

- `Gradient descent` (default) — Uses the Optimization Toolbox function `fmincon` to optimize the response signal subject to the constraints.

  The **Algorithm** options for `Gradient descent` are:

Algorithm Option	Learn More
`Sequential Quadratic Programming` (default)	"fmincon SQP Algorithm" in the Optimization Toolbox documentation.
`Active-Set`	"fmincon Active Set Algorithm" in the Optimization Toolbox documentation.
`Interior-Point`	"fmincon Interior Point Algorithm" in the Optimization Toolbox documentation.
`Trust-Region-Reflective`	"fmincon Trust Region Reflective Algorithm" in the Optimization Toolbox documentation.
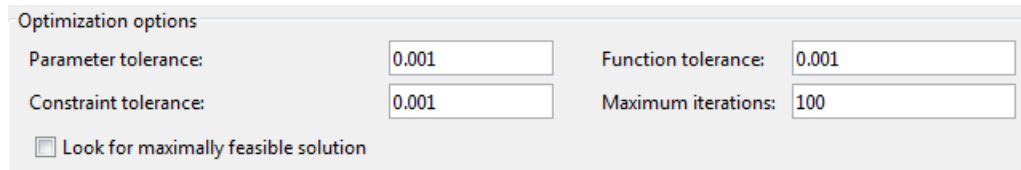
- `Pattern search` — Uses the Global Optimization Toolbox function `patternsearch`, an advanced direct search method, to optimize the response. This option requires the Global Optimization Toolbox.
- `Simplex search` — Uses the Optimization Toolbox function `fminsearch`, a direct search method, to optimize the response. `Simplex search` is most useful for simple problems and is sometimes faster than `Gradient descent` for models that contain discontinuities.

For more information on the problem formulations for each optimization method, see "How the Optimization Algorithm Formulates Minimization Problems" on page 3-3.

Selecting Optimization Termination Options

Use the **Optimization options** panel to specify when you want the optimization to terminate.

- **Parameter tolerance**: The optimization terminates when successive parameter values change by less than this number. For more details, refer to the discussion of the parameter `TolX` in the reference page for the Optimization Toolbox function `fmincon`.

- **Constraint tolerance**: This number determines the maximum limit by which the constraints can be violated, and still allow a successful convergence.

- **Function tolerance**: The optimization terminates when successive function values are less than this value. Changing the default **Function tolerance** value is only useful when you are tracking a reference signal or using the `Simplex search` method. For more details, refer to the discussion of the parameter `TolFun` in the reference page for the Optimization Toolbox function `fmincon`.

- **Maximum iterations**: The maximum number of iterations allowed. The optimization terminates when the number of iterations exceeds this number.

- **Look for maximally feasible solution**: When selected, the optimization continues after it has found an initial, feasible solution, until it finds a maximally feasible, optimal solution. When this option is unselected, the optimization terminates as soon as it finds a solution that satisfies the constraints and the resulting response signal sometimes lies very close to the constraint segment. In contrast, a maximally feasible solution is typically located further inside the constraint region.
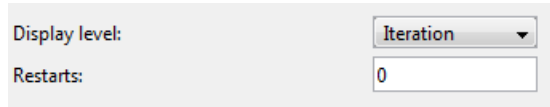
Note: If selected, the software ignores this option when tracking a reference signal.

By varying these parameters you can force the optimization to continue searching for a solution or to continue searching for a more accurate solution.

Selecting Additional Optimization Options

At the bottom of the **Optimization Options** panel is a group of additional optimization options.

Display level:	Iteration ▾
Restarts:	0

- "Display Level" on page 3-76
- "Restarts" on page 3-77

Display Level

The **Display level** option specifies the form of the output that appears in the Optimization Progress window. The options are:

- `Iterations` (default) — Displays information after each iteration
- `Off` — Turns off all output display
- `Notify` — Displays output only if the function does not converge
- `Final` — Displays only the final output

For more information on the type of iterative output that appears for the method you selected in **Method**, see the discussion of output for the corresponding function.

Method	Function	Output Information
Gradient descent	fmincon	fmincon section of "Function-Specific Headings" in the Optimization Toolbox documentation
Simplex search	fminsearch	fminsearch section of "Function-Specific Headings" in the Optimization Toolbox documentation
Pattern search	patternsearch	"Display to Command Window Options" in the Global Optimization Toolbox documentation

Restarts

In some optimizations, the Hessian may become ill-conditioned and the optimization does not converge. In these cases, it is sometimes useful to restart the optimization after it stops, using the endpoint of the previous optimization as the starting point for the next one. To automatically restart the optimization, indicate the number of times you want to restart in this field.

Create Linearization I/O Sets

This example shows how to create a linearization input/output set using the Response Optimization Tool.

You create a linearization input/output set using the Create Linearization I/O Set dialog box. This dialog box may be accessed in two ways:

- In the Response Optimization tool, select **Linearization I/Os** in the **New** drop-down list.

- In a requirement dialog box, in the **Select Systems to Bound** area, click [+].

Create Linearization I/O Set

To create a new linearization I/O set:

1 In your Simulink model, select one or more signals that you want to define as a linearization input or output point.

   The selected signals appear in the Create linearization I/O set dialog box under **Currently selected signals**.
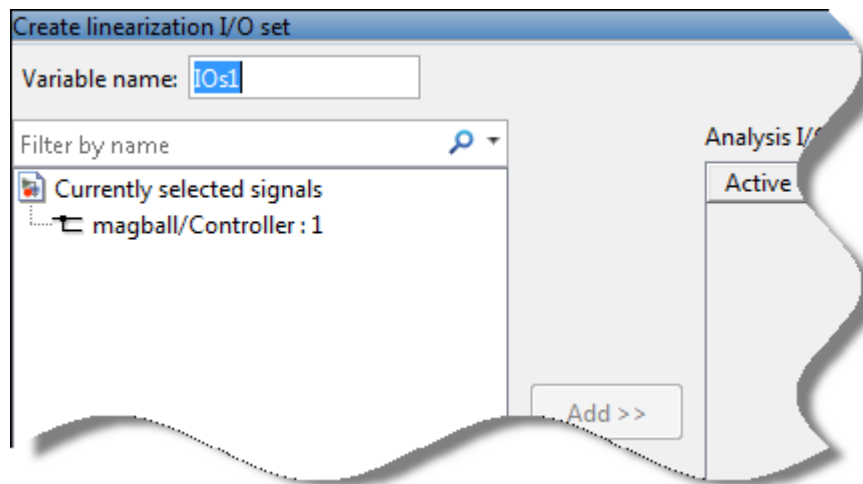
2 In the Create linearization I/O set dialog box, click a signal name under Currently selected signals.

3 Click **Add**. The signal appears in the list of Analysis I/Os.
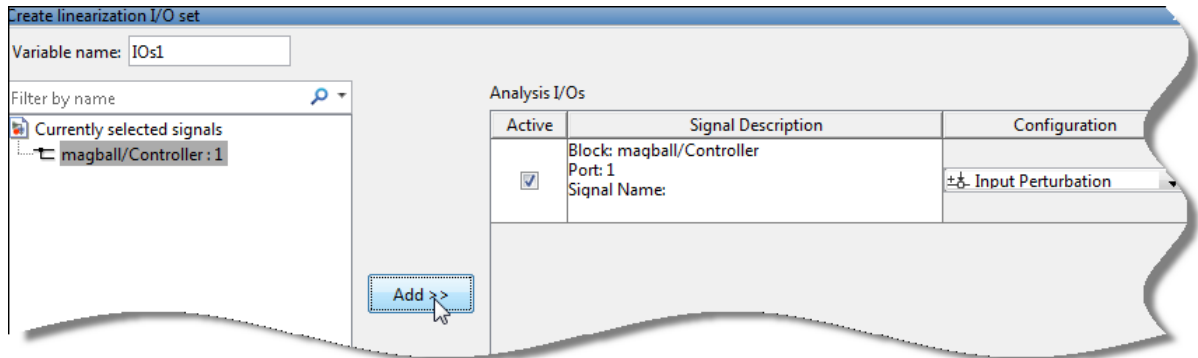
4 Select the linearization point type for a signal from the **Configuration** drop-down list for that signal. For example:

- If you want the selected signal to be a linearization output point, select `Output Measurement`.

- If you want the signal to be an open-loop output point, select `Open-loop Output`.

5 Repeat steps 1–4 for any other signals you want to define as linearization I/O points.

> **Tip** To highlight in the Simulink model the location of any signal in the current list of analysis I/O points, select the I/O point in the list and click **Highlight**.

6 After you define all the signals for the I/O set, enter a name for the I/O set in the **Variable name** field located at the top-left of the window.

7 Click **OK**.

Linearization Options

In this section...
"Accessing Linearization Options" on page 3-80
"Configuring Linearization Options" on page 3-80

Accessing Linearization Options

If you have Simulink Control Design, you can set several options for linearization. These options include the linearization methods and the sample time of the linear systems.

To set linearization options, click **Options** in the Response Optimization tool. A window opens. Select the **Linearization Options** tab.
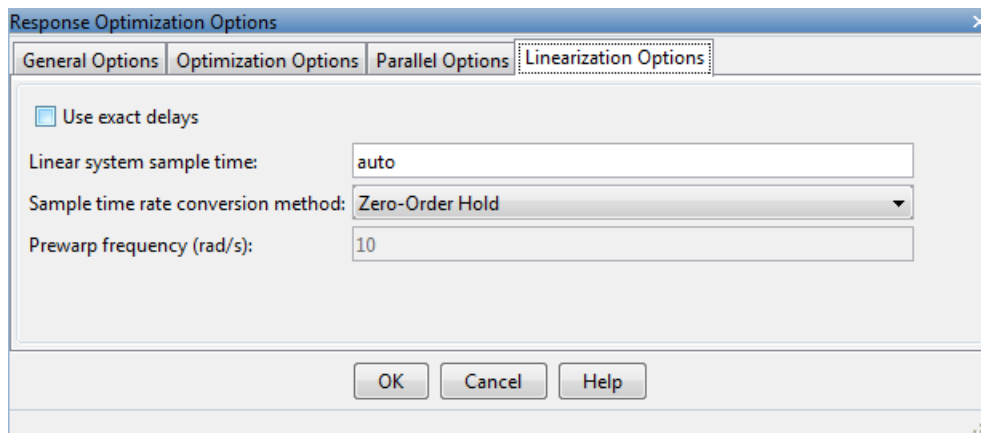
Configuring Linearization Options

Models with Time Delays

Simulink Control Design lets you choose whether to linearize models using exact representation or Pade approximation of continuous time delays. How you treat time delays during linearization depends on your nonlinear model.

Simulink blocks that model time delays are:

- Transport Delay block
- Variable Time Delay block
- Variable Transport Delay block
- Delay block
- Unit Delay block

By default, linearization uses Pade approximation for representing time delays in your linear model.

Use Pade approximation to represent time delays when:

- Applying more advanced control design techniques to your linear plant, such as LQR or H-infinity control design.
- Minimizing the time to compute a linear model.

Specify to linearize with exact time delays for:

- Minimizing errors that result from approximating time delays
- PID tuning or loop-shaping control design methods in Simulink Control Design
- Discrete-time models (to avoid introducing additional states to the model)

  The software treats discrete-time delays as internal delays in the linearized model. Such delays do not appear as additional states in the linearized model.

Specify How Delays are Treated

To specify whether the linearization should approximate delays or use them exactly, set the **Use exact delays** check box appropriately.

Linearization Sampling Time

To specify the sampling time of the linearized model, use the **Linear system sample time** box. By default, the software chooses the slowest applicable sampling time. Use 0 to specify a continuous-time linear model.

Linearization Rate Conversion Method

When you linearize models with multiple sample times, such as a discrete controller with a continuous plant, a rate conversion algorithm generates a single-rate linear model. The rate conversion algorithm affects linearization results.

Rate Conversion Method	When to Use
Zero-Order Hold	Use when you need exact discretization of continuous dynamics in the time-domain for staircase inputs.
Tustin	Use when you need good frequency-domain matching between a continuous-time system and the corresponding discretized system, or between an original system and the resampled system.
Tustin with Prewarping	Use when you need good frequency domain matching at a particular frequency between the continuous-time system and the corresponding discretized system, or between an original system and the resampled system.
Upsampling when possible (Zero-Order Hold, Tustin, and Tustin with Prewarping)	Upsample discrete states when possible to ensure gain and phase matching of upsampled dynamics. You can only upsample when the new sample time is an integer multiple of the sample time of the original system. Otherwise, the software uses an alternate rate conversion method.

Plots in the Response Optimization Tool

In this section...
"Adding Plots in Response Optimization Tool" on page 3-83
"Plotting Current Response" on page 3-83
"Plotting Intermediate Steps" on page 3-83
"Modifying Plot Properties" on page 3-83
"Plot Types" on page 3-85
"Export Design Variables and Requirement Values for an Iteration" on page 3-88

Adding Plots in Response Optimization Tool

To create a new plot or to add to an existing plot in the Response Optimization Tool, choose the variable to plot in the **Data to Plot** list. Then select the plot type using the **Add Plot** list. The **Add Plot** list has entries for the supported plot types for the given plot variable.

Plotting Current Response

To display the current response, click **Plot Model Response** in the **Response Optimization** tab of the Response Optimization tool. The current response appears as a thick line.

Plotting Intermediate Steps

To turn on, or off, the display of the response signal at intermediate steps during the optimization, right-click within the white space in the plot and select **Responses > Show Iteration Responses**. The response at an intermediate step is based on parameter values at that intermediate point in the optimization.

Modifying Plot Properties

Modifying Properties of Response Plots

Right-click the white space in a plot and select **Axes Properties** to open the Property Editor dialog box.

This figure shows the Property Editor dialog box for a step response.
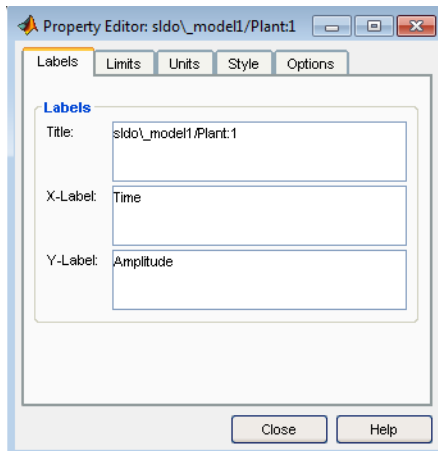
In general, you can change the following properties of response plots.

- **Labels** — Titles and X- and Y-labels

  To specify new text for plot titles and axis labels, type the new string in the field next to the label you want to change. The label changes immediately as you type, so you can see how the new text looks as you are typing.

- **Limits** — Numerical ranges of the x- and y- axes

  Default values for the axes limits make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the **Limits** pane fields. The **Auto-Scale** check box automatically clears if you click a different field. The new limits appear immediately in the response plot.

  To reestablish the default values, select the **Auto-Scale** check box again.

- **Units** — Units where applicable (e.g., rad/s to Hertz). If you cannot customize units, the Property Editor displays that no units are available for the selected plot.

- **Style** — Show a grid, adjust font properties, such as font size, bold and italics, and change the axes foreground color

- **Options** — Change options where applicable. These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

Plot Types

Response Plots

You can view model signals and the requirements applied to the signal using a response plot. You can also plot the frequency response of a system (requires Simulink Control Design).

The response plot shows the system response as it varies during optimization. You can also view the uncertain system responses in the plot.
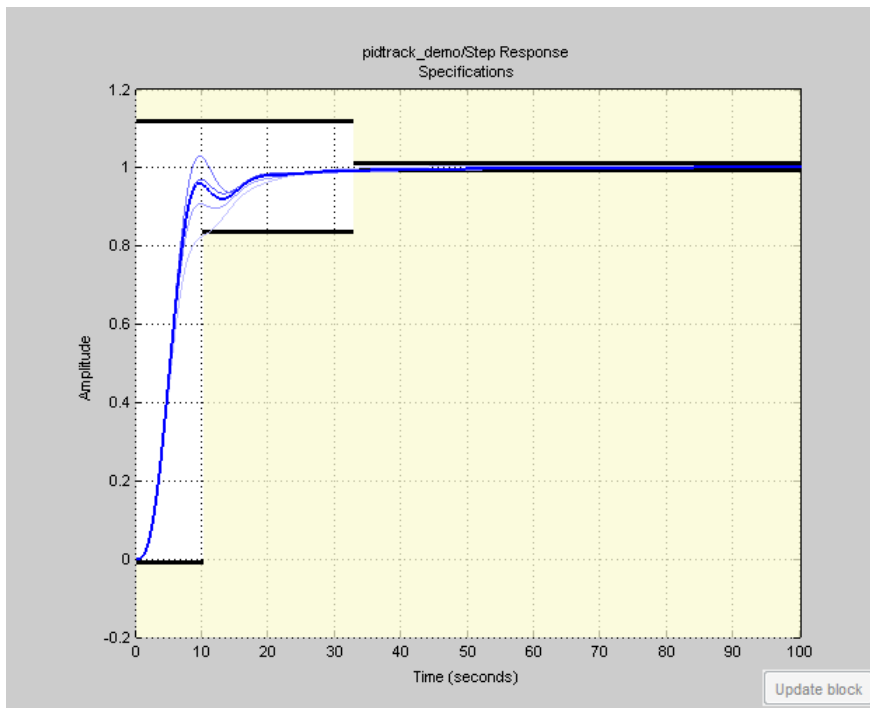
Tip To select the responses and systems displayed in a given plot, right-click on the plot and use the **Systems** and **Responses** menu.

Spider Plots

You can compare the values of design variable sets or evaluated requirements using a spider plot.

Spider plots depict multivariate data using an axis for each variable. The various axes are arranged clockwise and have a common intersecting point, as this example shows:
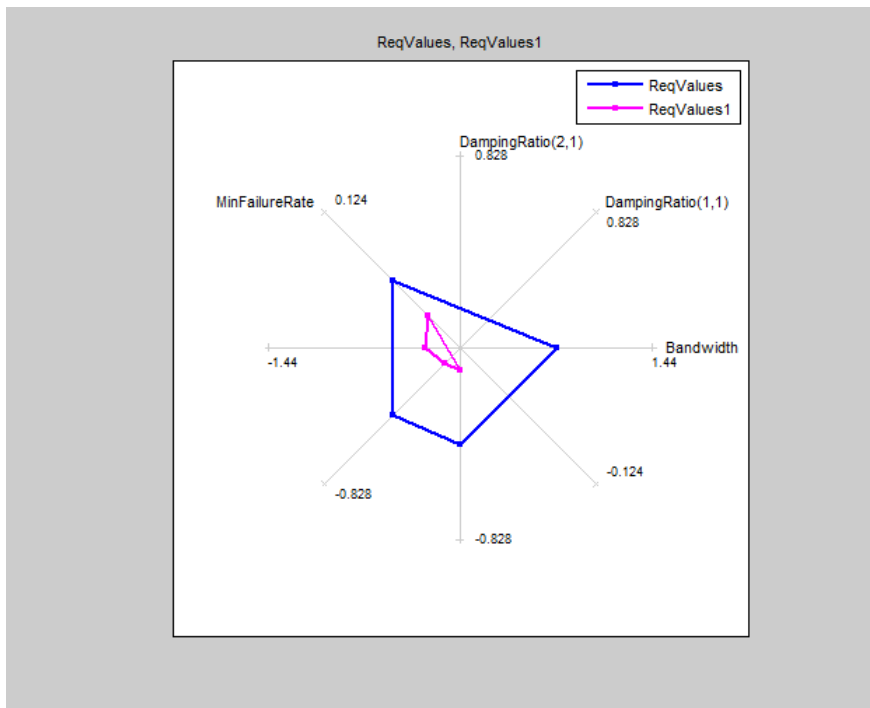
Tip To view only some of the requirement values in a given plot, right-click on the plot and select the requirements in the **Show** list.

For information on using a spider plot to compare design variables sets or evaluated requirements, see "Compare Requirements and Design Variables Using Spider Plot" on page 3-89

Iteration Plots

You can plot the values of design variables and requirements as they vary during optimization using an iteration plot.

Iteration plots depict the value of the plot variable(s) for each iteration. The x-axis represents the iteration number, as this example shows:
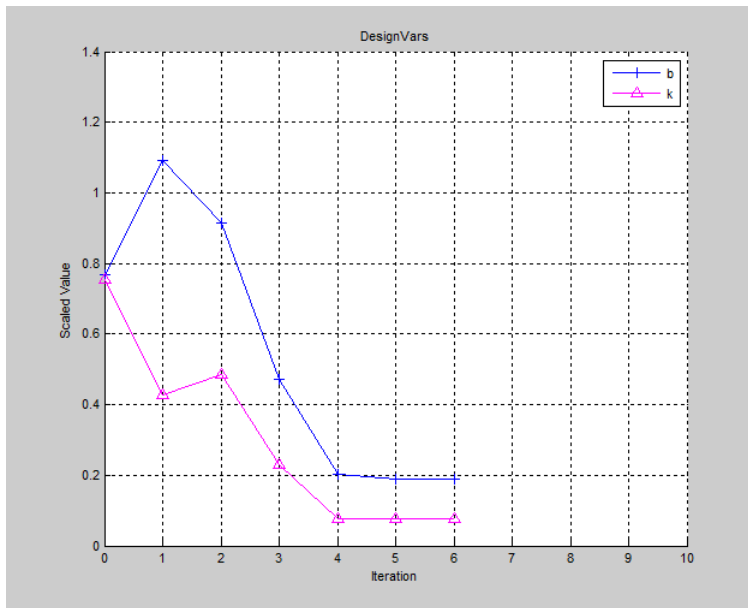
You can export the values of a plotted variable for a given iteration. For more information, see "Export Design Variables and Requirement Values for an Iteration" on page 3-88.

Tip To view scaled values of the plotted variable(s), right-click on the plot and select **Show scaled values**.

Export Design Variables and Requirement Values for an Iteration

To export the values of design variables or requirements:

1 Open the Export Iteration Data dialog box.

Right-click on the iteration plot and select **Export**.

2 Specify the plotted variable that you want to export using the **Data to export** list.

3 Specify the iteration for which you want to export data in the **Iteration(s) to export** box.

To specify multiple iterations, use a vector of integers. For example, `[0 2 5]`.

4 Specify the variable name for the exported data using the **Export to a variable named** box.

5 Export the data to the **Data** area of the tool.

Click **OK**. The exported data variable appears in the **Data** area.

Note: The iteration number is added as a suffix to the exported data variable name.

Compare Requirements and Design Variables Using Spider Plot

This example shows how to use a spider plot to compare requirement evaluations before and after optimizing the response. You can use a similar procedure to compare the values of sets of design variables.

Open the Simulink model and load the pre-configured Response Optimization Tool session.

For this example, which uses a distillation column model, the step response requirements are preconfigured and loaded in the model workspace.

1 Open the distillation model.

```
sys = 'distillation_demo';
open_system(sys)
```

2 Open the Response Optimization Tool.

In the Simulink model window, select **Analysis** > **Response Optimization**.

Alternatively, click the `Response Optimization GUI with preloaded data` block in the model and skip the next step.

3 Load the preconfigured Response Optimization Tool session.

Click the **Response Optimization** tab. In the **Open Session** drop-down list, select **Open from model workspace**. A window opens where you select the Response Optimization Tool session to load. Select `distillation_optim` and click **OK**.

The preconfigured step response requirements are loaded in the Response Optimization Tool.

Evaluate the requirement before optimization.

In the **Response Optimization** tab, click **Evaluate Requirements**.

A new variable, `ReqValues`, containing the evaluation of the requirements appears in the **Data** area.

When optimizing the model response, you create a set of requirements that it must satisfy. If the requirements are violated, meaning that they evaluate to nonnegative values, the design variables must be optimized. After the optimization, you can compare the original requirement value with the requirement evaluated using the optimized design variable values.

Plot the requirement value before optimization.

1 In the **Data to Plot** list, select `ReqValues`.

2 In the **Add Plot** list, select `Spider plot`.
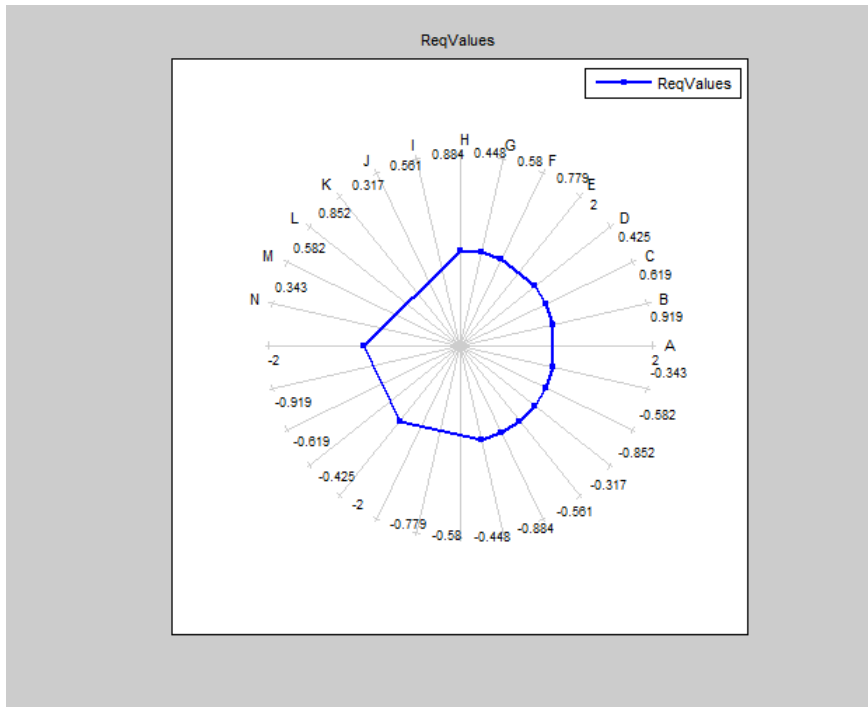
The plot has an axis for each edge-and-signal combination defined in the `distillation_demo/Desired Step Response` check block. Points on each axis represent the violation for that signal-edge combination and the plot connects these points to form a closed polygon representing the initial design. Note that some points are negative, representing satisfied constraints, and some positive, representing violated constraints.

Optimize the model.

Click **Optimize**.

A new variable, `ReqValues1`, containing the evaluation of the requirements using the optimized design variables appears in the **Data** area.

Compare the requirement values before and after optimization.

1   In the **Data to Plot** list, select `ReqValues1`.

2   In the **Add Plot** list, select `Spider plot 1`.
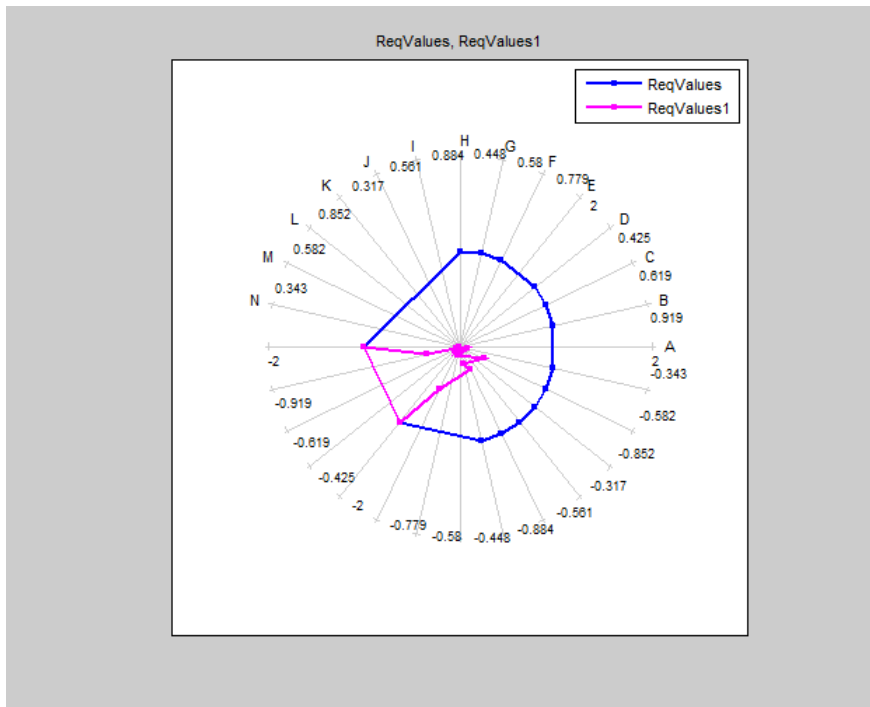
The optimized requirement values, `ReqValues1`, are all negative or zero, indicating that all the constraints are satisfied.

More About

·   "Export Design Variable Values for Specific Iteration" on page 3-92

Export Design Variable Values for Specific Iteration

This example shows how to export the design variable values for a specific optimization iteration.

During optimization, the optimization solver simulates the model using a different set of design variables at each iteration. After the optimization completes, you can export the values for an iteration from the iteration plot of the design variable set.

For this example, load a preconfigured Response Optimization tool session. Optimize the model, and export the design variable set values for the third iteration.

Open the Simulink model and load the preconfigured Response Optimization tool session.

```
load('distillation_demo_sdosession_export_iter_dv.mat')
sdotool(SDOSessionData)
```

The Response Optimization tool opens and loads the preconfigured session. **Iteration Plot 1** is configured to plot the values of `DesignVars` for each optimization iteration.

Click **Optimize**.

The optimization completes after four iterations.

Select the iteration plot of the design variable set.

Click **Iteration plot 1**.

Open the Export Iteration Data dialog box.

Right-click on the iteration plot, and select **Export**.
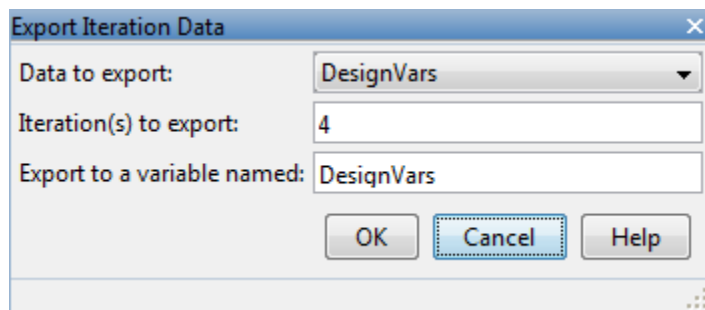
Specify details regarding exporting the design variable set data:

- In the **Data to export** list, select `DesignVars`.
- In the **Iteration(s) to export** box, enter `3`.

  To specify multiple iterations, use a vector of integers. For example, `[0 2 5]`.
- In the **Export to a variable named** box, enter `DesignVars_iter`.

Export the design variable values set to the **Data** area of the tool.

Click **OK**. The exported data variable, `DesignVars_iter_3`, appears in the **Data**.

Note: You will see the iteration number suffixed to the exported data variable name.

Related Examples
- "Update Model with Design Variables Set" on page 3-67
- "Compare Requirements and Design Variables Using Spider Plot" on page 3-89

More About
- "Iteration Plots" on page 3-87

Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)

This example shows how to tune a controller to satisfy time- and frequency-domain design requirements using the Response Optimization tool.

The example requires Simulink® Control Design™.

Aircraft Longitudinal Flight Control Model

Open the Simulink model.

```
sys = 'sdoAircraft';
open_system(sys);
```
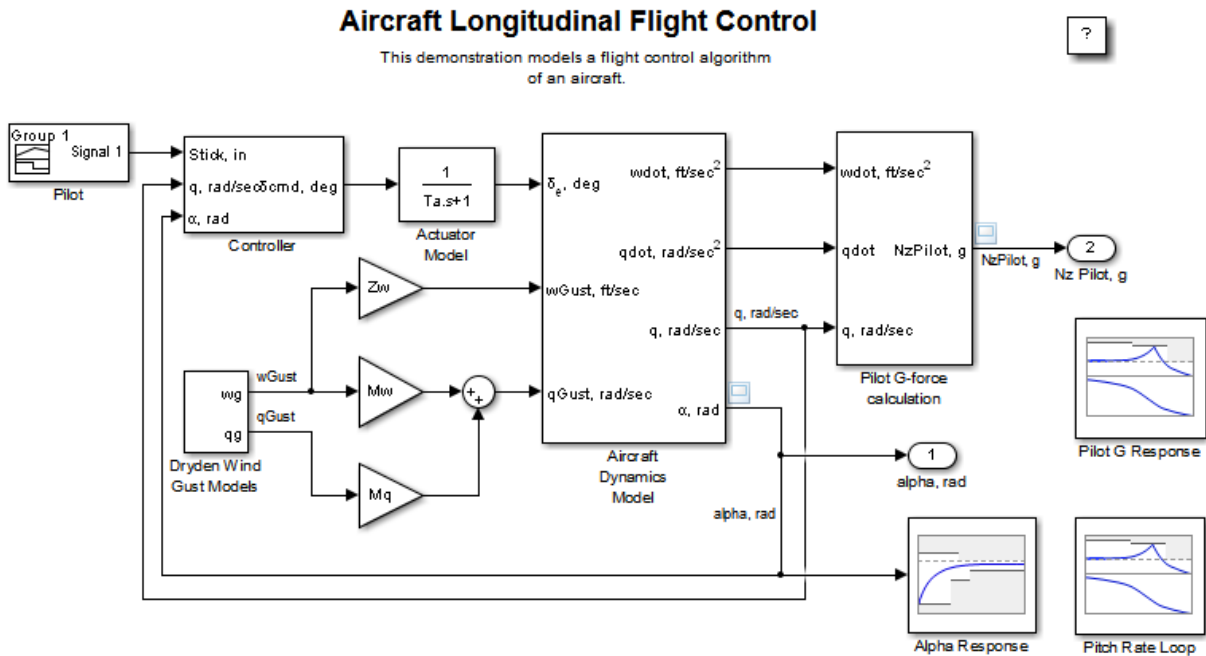
The aircraft model is based on the Simulink `slexAircraftExample` model. The model includes:

- Subsystems to model aircraft dynamics (`Aircraft Dynamics Model`), wind gusts (`Dryden Wind Gust Models`), and pilot G-forces (`Pilot G-force calculation`).

- A step change applied to the aircraft joystick at 1 second into the simulation that causes the aircraft to pitch upward.

Controller Design Problem

You tune the controller gains to meet the following time- and frequency-domain design requirements:

- Angle-of-attack `alpha` response to a step change in the joystick has a rise time of less than 1 second, less than 1% overshoot, and settles to within 1% of steady state within less than 5 seconds

- Pitch-rate control loop has good tracking below 1 rad/s and 20 dB noise rejection above 100 rad/s

- Closed loop response from joystick to pilot G-Force is below 0 dB above 5 rad/s.

These requirements reduce the high frequency G-forces experienced by the pilot in response to joystick changes while still maintaining flight performance.

The model includes the following blocks (from Simulink® Design Optimization™ and Simulink Control Design Model Verification libraries):

- `Alpha Response` specifies the alpha step response requirement.

Check Step Response Characteristics

Assert that the input signal satisfies bounds specified by step response characteristics.

Bounds | **Assertion**

☑ Include step response bound in assertion

Step time (seconds): | 1

Initial value: | 0 | Final value: | 0.5

Rise time (seconds): | 1 | % Rise: | 80

Settling time (seconds): | 5 | % Settling: | 1

% Overshoot: | 1 | % Undershoot: | 1

☑ Enable zero-crossing detection

Show Plot | ☐ Show plot on block open | Response Optimization...

OK | Cancel | Help | Apply

- `Pitch Rate Loop` specifies the pitch-rate performance requirement.

The linearization inputs/outputs are already selected in the **Linearizations** tab. The pitch-rate loop starts from the input of the controller (the controller error signal) and ends at the output of the pitch-rate sensor. The angle-of-attack loop is opened signal so that the block only computes the pitch-rate loop response. The linear system is computed at a simulation time of 0.

Bode Plot

Compute and display a linear system on a Bode plot. You can also specify bounds on the linear system and assert that the bounds are satisfied.

| Linearizations | Bounds | Logging | Assertion |

Linearization inputs/outputs:

Block : Port : Bus Element	Configuration
sdoAircraft/Controller/error : 1	Input Perturbation
sdoAircraft/Controller/Kq : 1	Open-loop Output
sdoAircraft/Controller/Ka : 1	Loop Break

+

×

Linearize on: Simulation snapshots ▼

Snapshot times: [0]

Trigger type: Rising edge ▼

▶ Algorithm Options

▶ Labels

Show Plot   ☐ Show plot on block open   Response Optimization...

OK   Cancel   Help   Apply

The **Bounds** tab specifies the following pitch-rate loop shape requirements:

- Greater than 20 dB over the range 0.01 rad/s to 0.1 rad/s

- Greater than 0 dB over the range 0.1 rad/s to 1 rad/s

- Less than -20 dB over the range 100 rad/s to 1000 rad/s

Bode Plot

Compute and display a linear system on a Bode plot. You can also specify bounds on the linear system and assert that the bounds are satisfied.

| Linearizations | Bounds | Logging | Assertion |

☐ Include upper magnitude bound in assertion

Frequencies (rad/s): [100 1000]

Magnitudes (dB):   [-20 -20]

☑ Include lower magnitude bound in assertion

Frequencies (rad/s): [0.01 0.1; 0.1 1]

Magnitudes (dB):   [20 20; 0 0]

Show Plot ☐ Show plot on block open   Response Optimization...

OK   Cancel   Help   Apply

- `Pilot G Response` specifies the G-force requirement.

The linearization inputs/outputs are already selected in the **Linearizations** tab. The linear system is computed at a simulation time of 0.

Bode Plot

Compute and display a linear system on a Bode plot. You can also specify bounds on the linear system and assert that the bounds are satisfied.

| Linearizations | Bounds | Logging | Assertion |

Linearization inputs/outputs:

Block : Port : Bus Element	Configuration
sdoAircraft/Pilot : 1	Input Perturbation
sdoAircraft/Pilot G-force calculati...	Output Measurement

+

×

Linearize on: Simulation snapshots ▼

Snapshot times: [0]

Trigger type: Rising edge ▼

▶ Algorithm Options

▶ Labels

Show Plot   ☐ Show plot on block open   Response Optimization...

⊙   OK   Cancel   Help   Apply

The **Bounds** tab specifies the G-force requirements of less than 0 dB over the range 5 rad/s 100 rad/s.

Bode Plot

Compute and display a linear system on a Bode plot. You can also specify bounds on the linear system and assert that the bounds are satisfied.

| Linearizations | Bounds | Logging | Assertion |

☑ Include upper magnitude bound in assertion

Frequencies (rad/s): [5 100]

Magnitudes (dB): [0 0]

☐ Include lower magnitude bound in assertion

Frequencies (rad/s): []

Magnitudes (dB): []

Show Plot   ☐ Show plot on block open   Response Optimization...

OK   Cancel   Help   Apply

Open the Response Optimization Tool

Open the Response Optimization tool to configure and run design optimization problems interactively. Click **Response Optimization** on the Block Parameters dialog of `Alpha Response`, `Pitch Rate Loop` or `Pilot G Response` block. Alternatively, type `sdotool('sdoAircraft')`. To show multiple requirement plots at the same time, use the **View** tab in the tool.
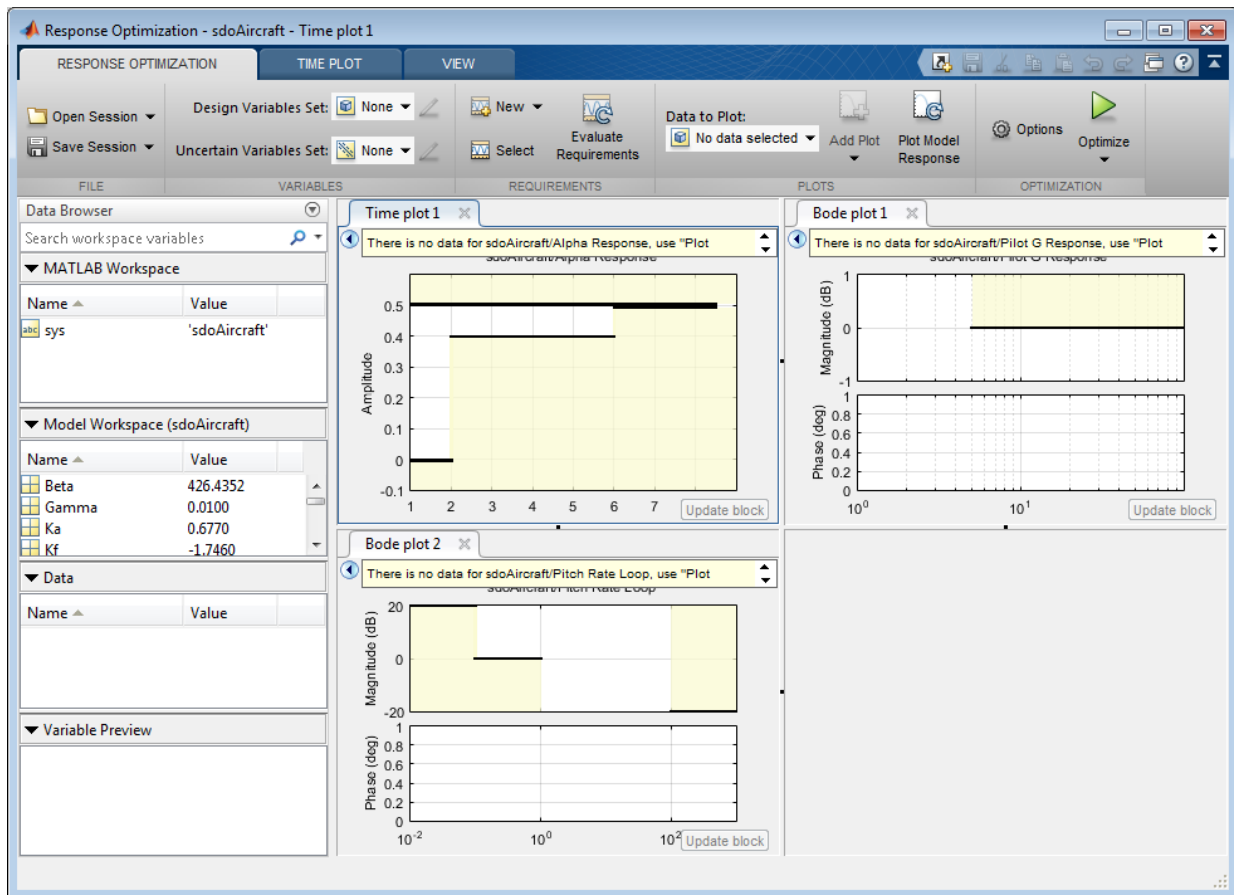
The tool detects the requirements specified in the Model Verification blocks and automatically includes them as requirements to satisfy.

Specify Design Variables

Specify the following model parameters as design variables for optimization:

- Controller gains `Ki` and `Kf`

- Pitch-rate sensor gain `Kq`

- Alpha sensor gain `Ka`

In the **Design Variables Set** drop-down list, select **New**. A dialog to select model parameters for optimization opens.

Select `Ki`, `Kf`, `Kq` and `Ka`. Click << to add the selected parameters to the design variables set.

Specify minimum and maximum gain values, the `Ki` and `Kf` values must remain negative while `Ka` and `Kq` must remain positive.

Press **Enter** after you enter the values.

	Variable	Value	Minimum	Maximum	Scale
☑	Ka	0.677000...	0.001	Inf	1
☑	Kf	-1.746	-Inf	-0.001	2
☑	Ki	-3.864	-Inf	-0.001	4
☑	Kq	0.8156	0.001	Inf	1
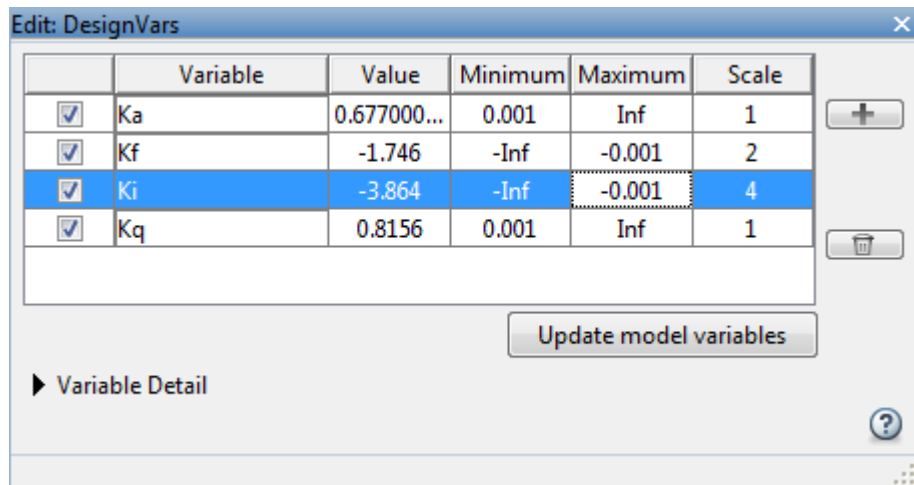
Click **OK**. A new variable `DesignVars` appears in the **Response Optimization Tool Workspace**.

Evaluate the Initial Design

Click **Plot Model Response** to simulate the model and check how well the initial design satisfies the design requirements.
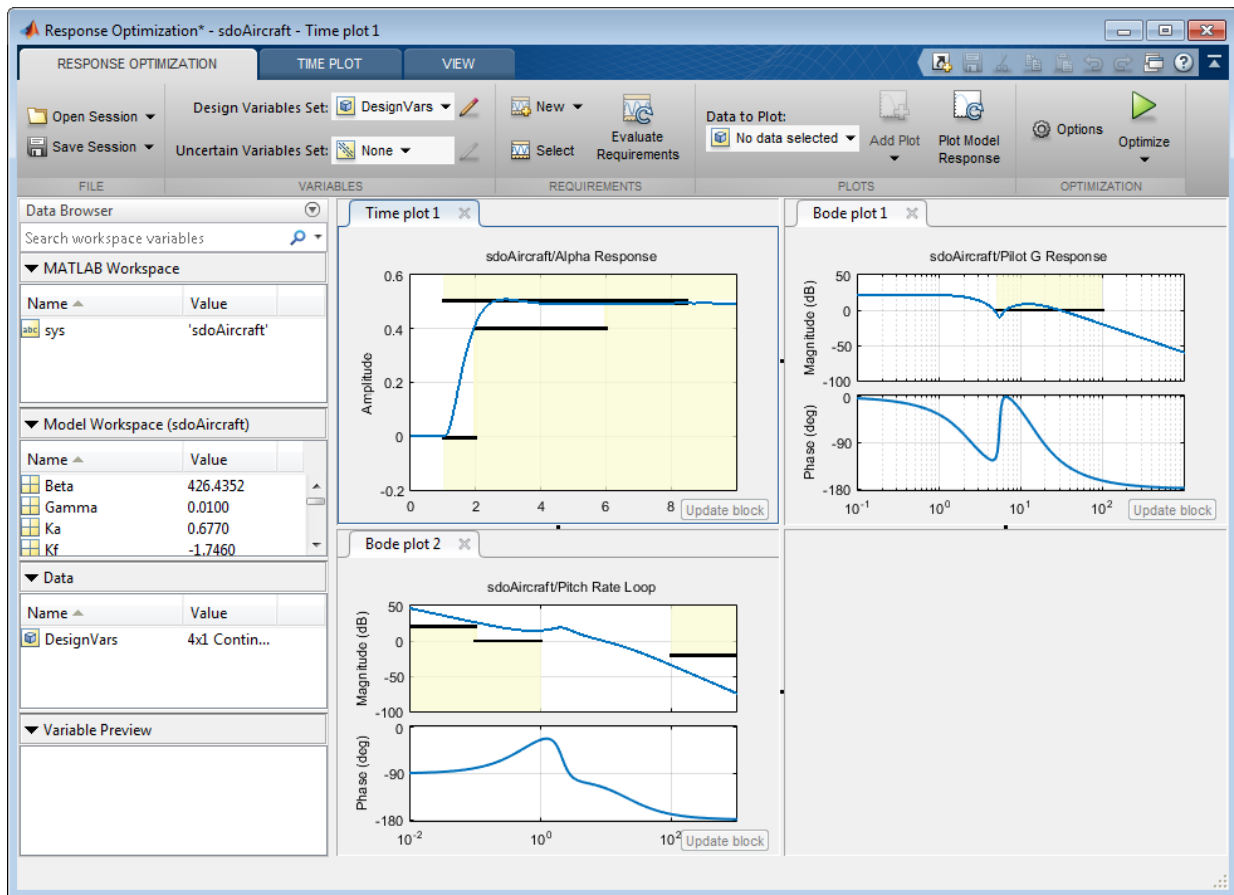
The plots indicate that the current design does not satisfy the pilot G-force requirement and the alpha step response overshoot requirement is violated.

Optimize the Design

Create a plot to display how the controller variables are modified during the optimization. In the **Data To Plot** drop-down list, select **DesignVars**, which contains

the optimization design variables `Ki`, `Kf`, `Kq` and `Ka`. In the **Add Plot** drop-down list, select **Iteration plot**.
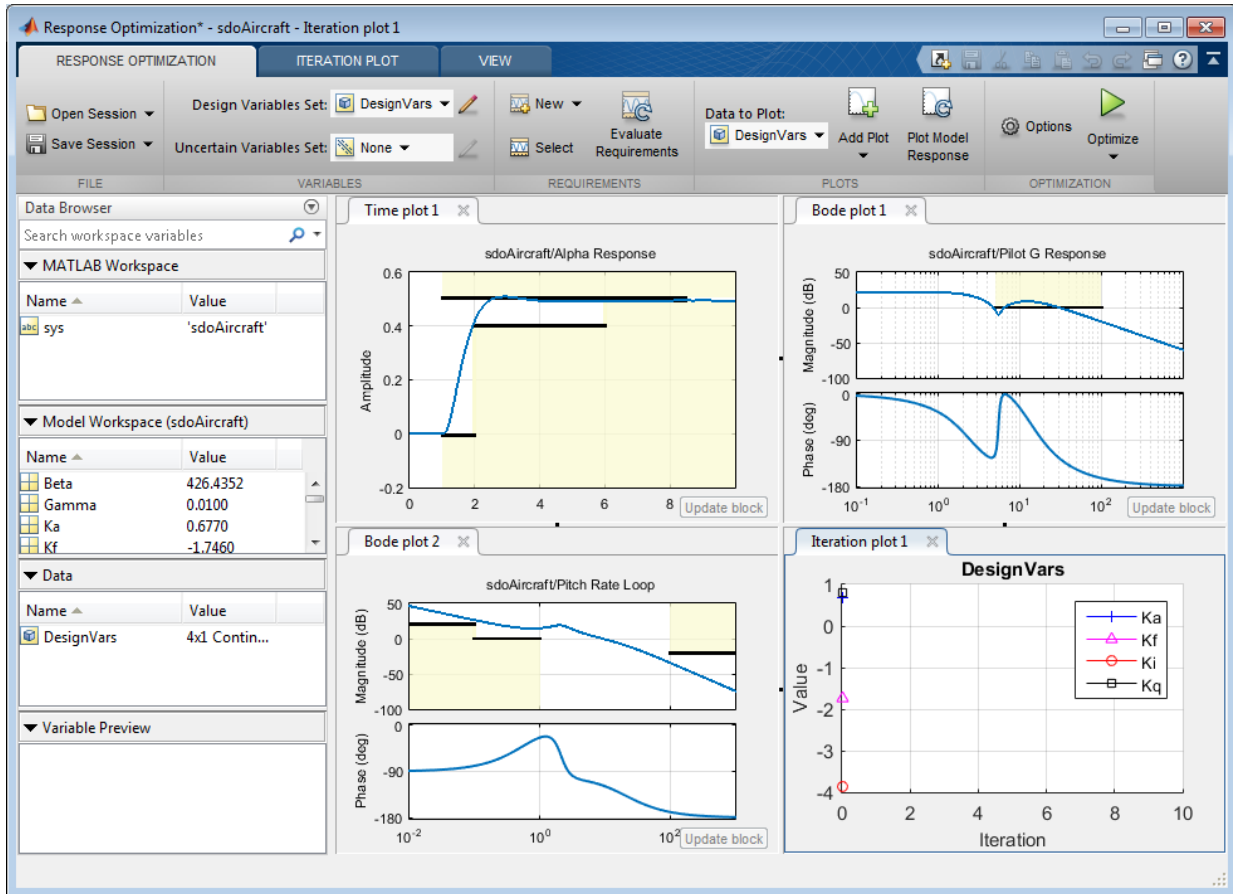
Click **Optimize**.
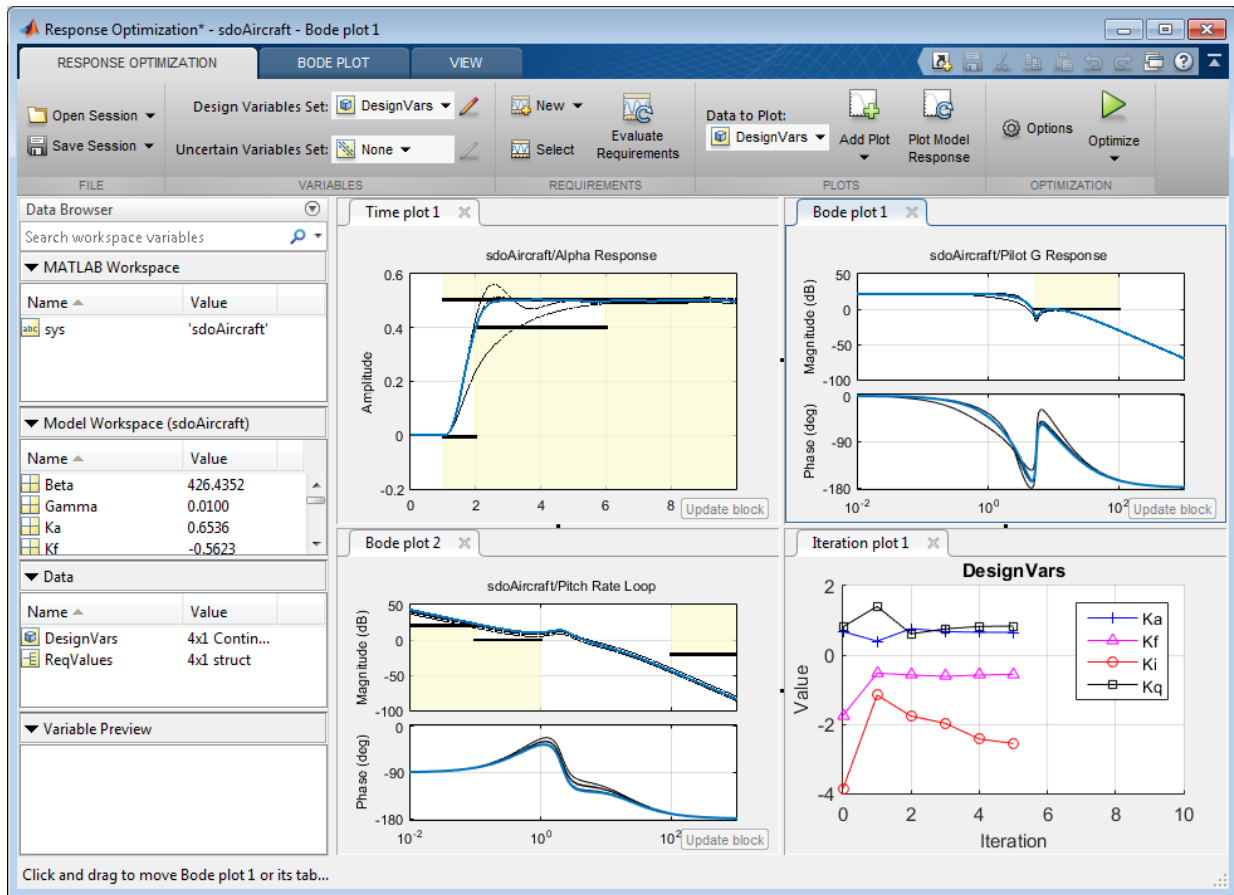
Iteration	F-count	Alpha Response (... (<=0)	Pilot G Response ... (<=0)	Pitch Rate Loop (... (<=0)	Pitch Rate Loop (... (>=0)
0	9	0.1450	8.5841	-0.7160	0.3122
1	20	0.4060	-1.7578	-1.0067	0.0215
2	31	0.1081	-1.0047	-1.3244	-0.1505
3	41	0.2407	-0.0478	-1.2014	-0.0129
4	51	0.0067	0.0584	-1.1881	0.1180
5	61	2.2579e-04	0.0034	-1.1965	0.1432

Optimization started 01-Oct-2012 11:15:57

Optimization converged, 01-Oct-2012 11:16:39

Optimized variable values written to 'DesignVars' in the Design Optimization workspace

Save Iteration...   Display Options...   Optimize

To load a pre-configured file and run the optimization, click **Open** in the **Response Optimization** tab and select sdoAircraft_sdosession.mat. Alternatively load the project by typing:

>> load sdoAircraft_sdosession

>> sdotool(SDOSessionData)

The optimization progress window updates at each iteration and shows that the optimization converged after 5 iterations.

The Alpha Response and Pilot G Response plots indicate that the design requirements are satisfied. The DesignVars plot shows that the controller gains converged to new values.

To view the optimized design variable values, click `DesignVars` in the **Response Optimization Tool Workspace**. The optimized values of the design variables are automatically updated in the Simulink model.

```
% Close the model
bdclose('sdoAircraft')
```

Design Optimization to Meet a Custom Objective (GUI)

This example shows how to optimize a design to meet a custom objective using the Response Optimization tool. You optimize the cylinder parameters to minimize the cylinder geometry and satisfy design requirements.

Hydraulic Cylinder Model

The hydraulic cylinder model is based on the Simulink model `sldemo_hydcyl`. The model includes:

- `Pump` and `Cylinder Assembly` subsystems. For more information on the subsystems, see "Single Hydraulic Cylinder Simulation".

- A step change applied to the cylinder control valve orifice area that causes the cylinder piston position to change.

Hydraulic Cylinder Design Problem

You tune the cylinder cross-sectional area and piston spring constant to meet the following design requirements:

- Ensure that the piston position has a step response rise time of less than 0.04 seconds and setting time of less than 0.05 seconds.

- Limit the maximum cylinder pressures to 1.75e6 N/m.

- Minimize the cylinder cross-sectional area.

Open the Response Optimization Tool

Open the Response Optimization tool to configure and run design optimization problems interactively.

```
sdotool('sdoHydraulicCylinder')
```

Single Hydraulic Cylinder Simulation
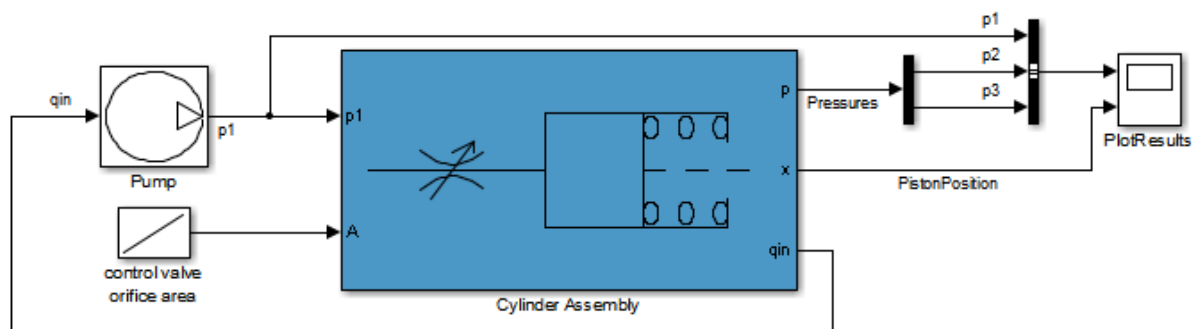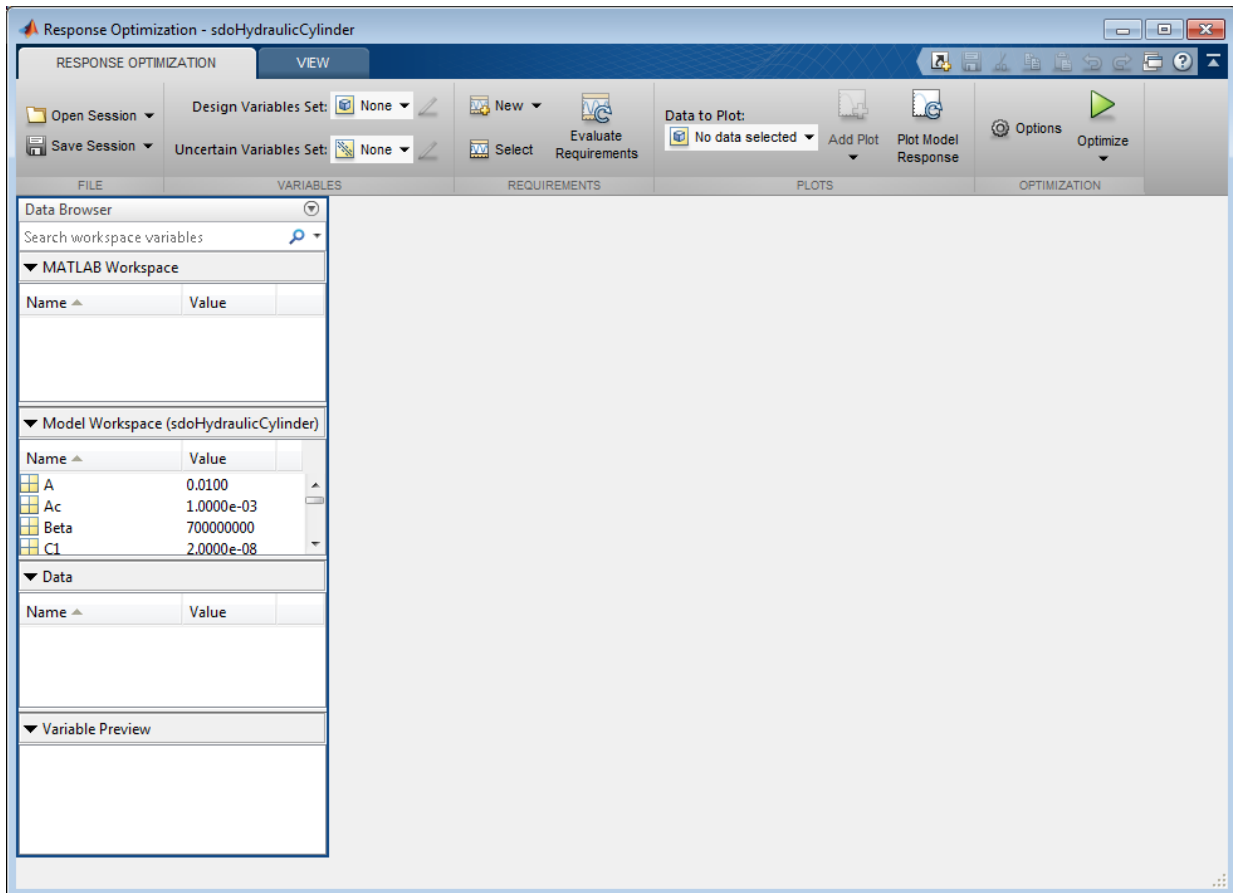
Copyright 1990-2011 The MathWorks, Inc.

Specify Design Variables

Specify the following model parameters as design variables for optimization:

- Cylinder cross-sectional area Ac

- Piston spring constant K

In the **Design Variables Set** drop-down list, select **New**. A dialog to select model parameters for optimization opens.
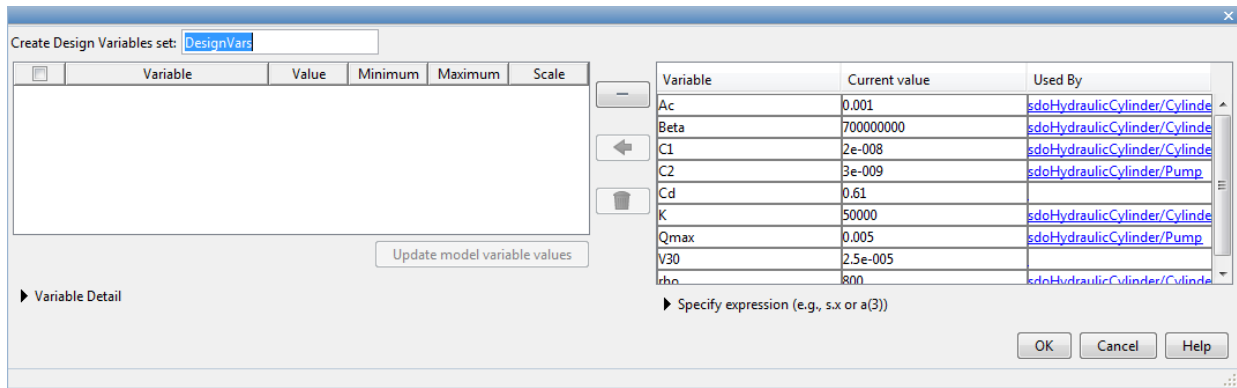
Select `Ac` and `K`. Click <- to add the selected parameters to the design variables set.

Limit the cylinder cross-sectional area to circular area with radius between 1 and 2 centimeters and the piston spring constant to a range of 1e4 to 10e4 N/m. To do so, specify the maximum and minimum for the corresponding variable in the **Maximum** and **Minimum** columns.

Because the variable values are different orders of magnitude, scale `Ac` by 1e-3 and `K` by 1e5.

Press **Enter** after you specify the values.
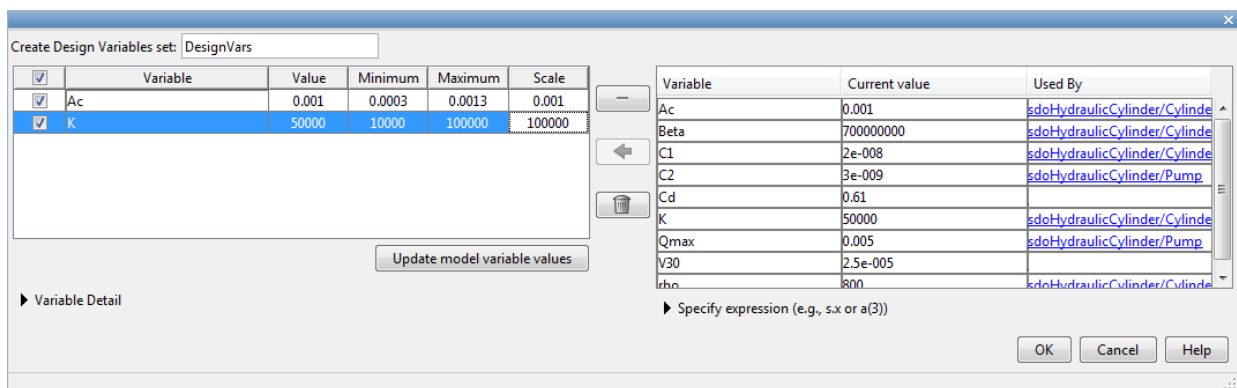
Click **OK**. A new variable `DesignVars` appears in the **Response Optimization Tool Workspace**.

Specify Design Requirements

The design requirements require logged model signals. During optimization, the model is simulated using the current value of the design variables and the logged signal is used to evaluate the design requirements.

Log the cylinder pressures, which is the first output port of the `Cylinder Assembly` block.

In the **New** drop-down list, select **Signal**. A dialog to select model signals to log opens.
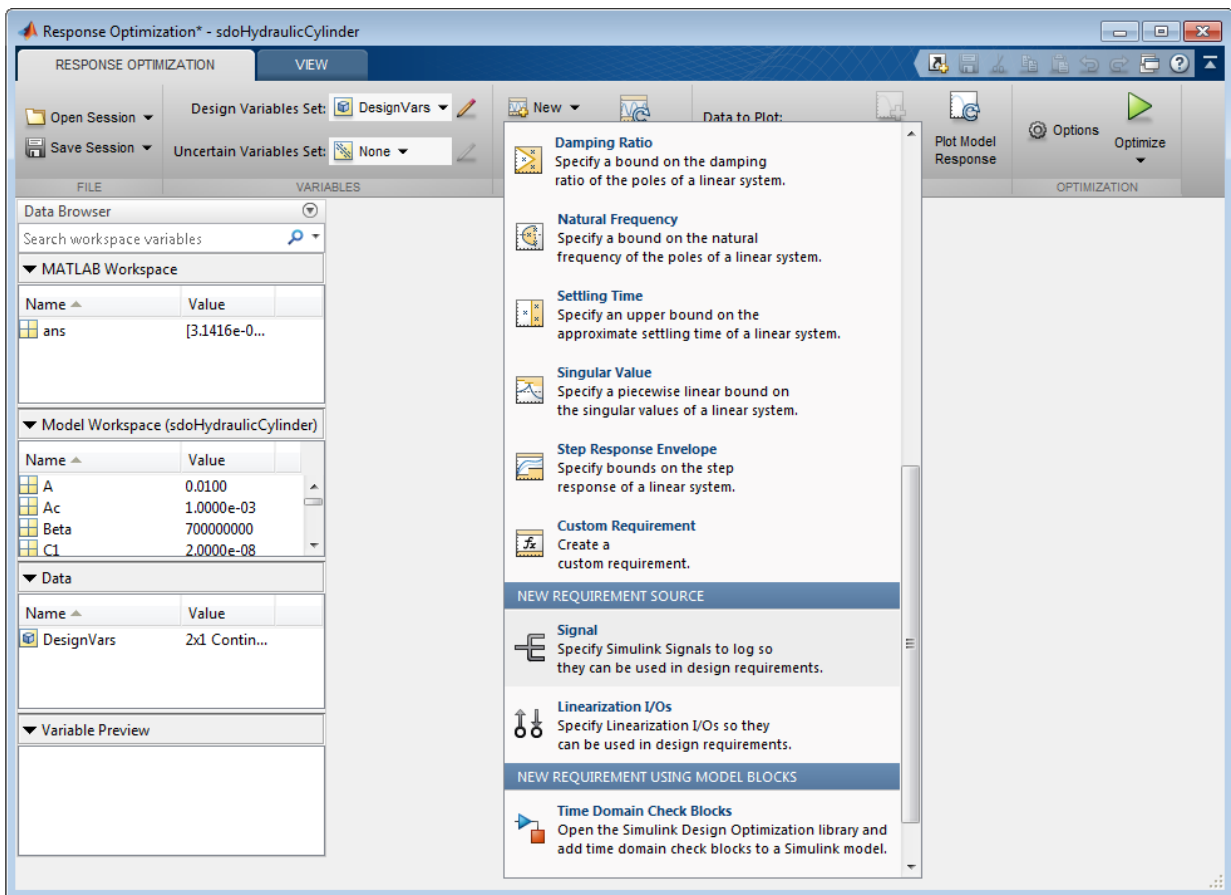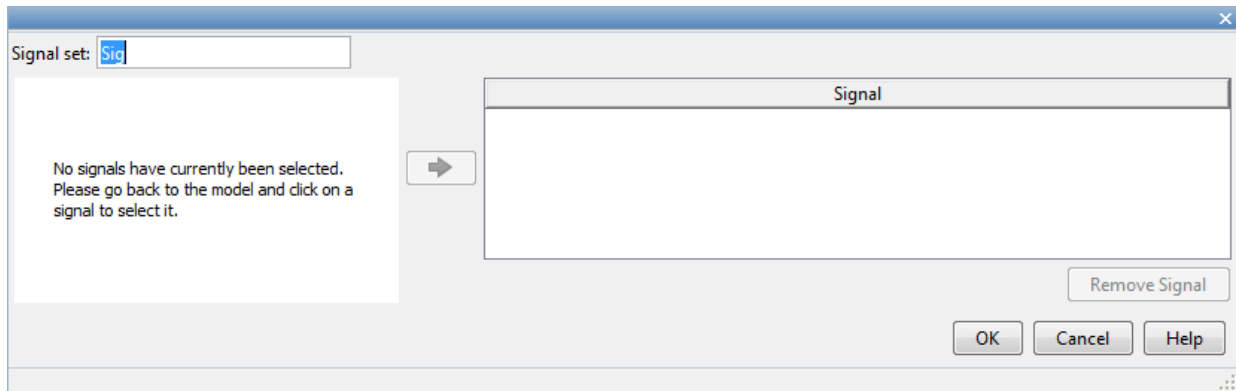
Enter `Pressures` as the signal name in the **Signal set** field. Then, in the Simulink model, click the first output port of the `Cylinder Assembly` block named `Pressure`. The dialog updates to display the selected signal.

Select the signal in the dialog and click -> to add it to the signal set.
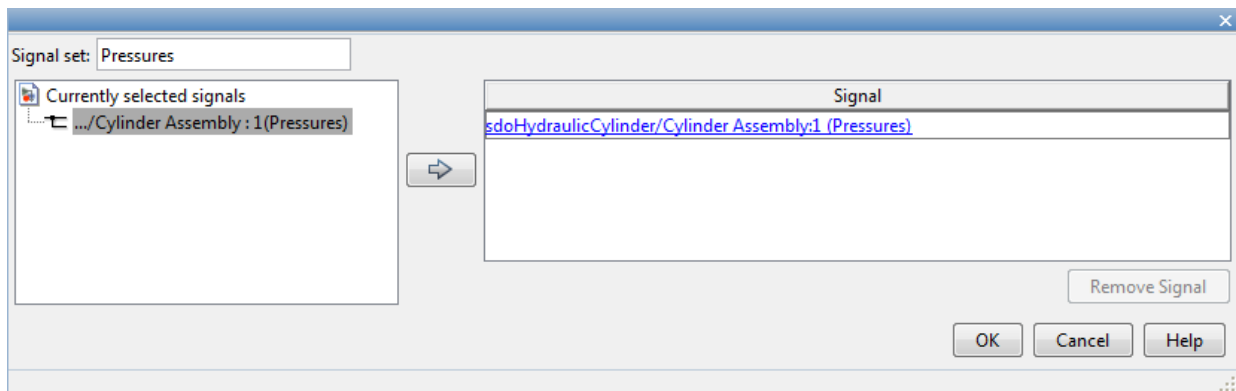
Click **OK**. A new variable **Pressures** appears in the **Response Optimization Tool Workspace**.

Similarly, log the piston position, which is the second output of the `Cylinder Assembly` block, in a variable named `PistonPosition`.
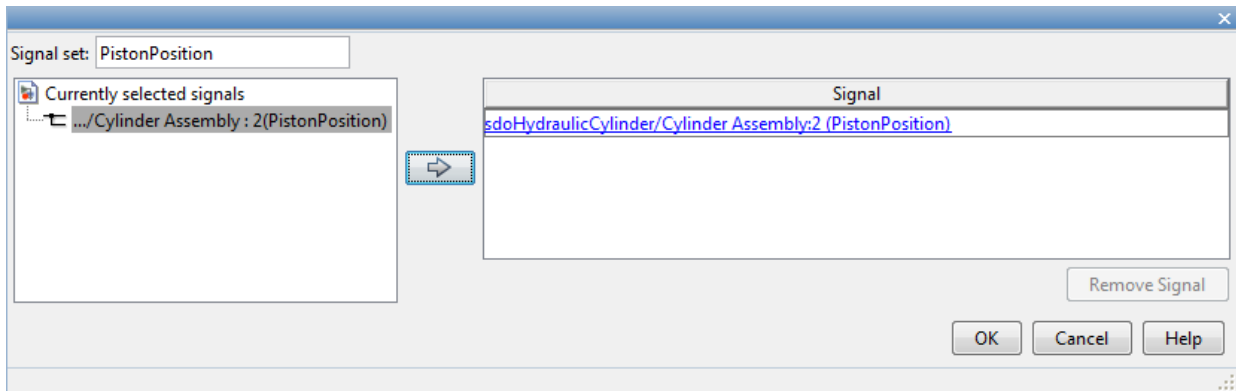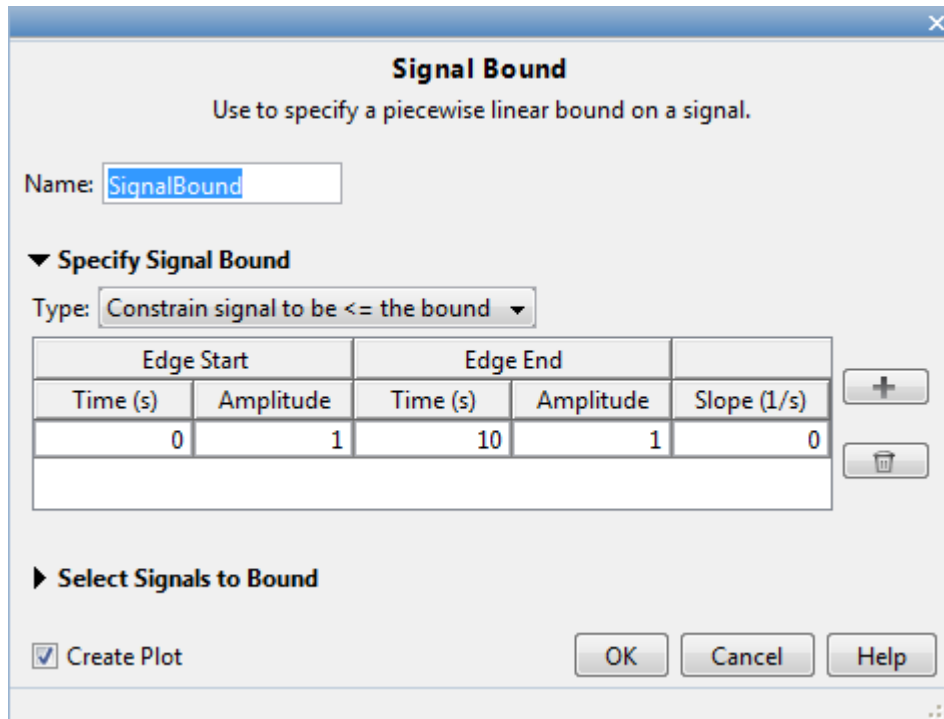
Specify the maximum cylinder pressure requirement of less than 1.75e6 N/m.

In the **New** drop-down list, select **Signal Bound**. A dialog to create a signal bound requirement opens.

Designate the **Requirement Name** as MaxPressure. In both the start and end
Amplitude columns, enter the maximum pressure requirement of 1.75e6 N/m, and set
the **Edge End Time** to 0.1 s. In the **Select Signals to Bound** area, select Pressures,
the signal on which this requirement applies.
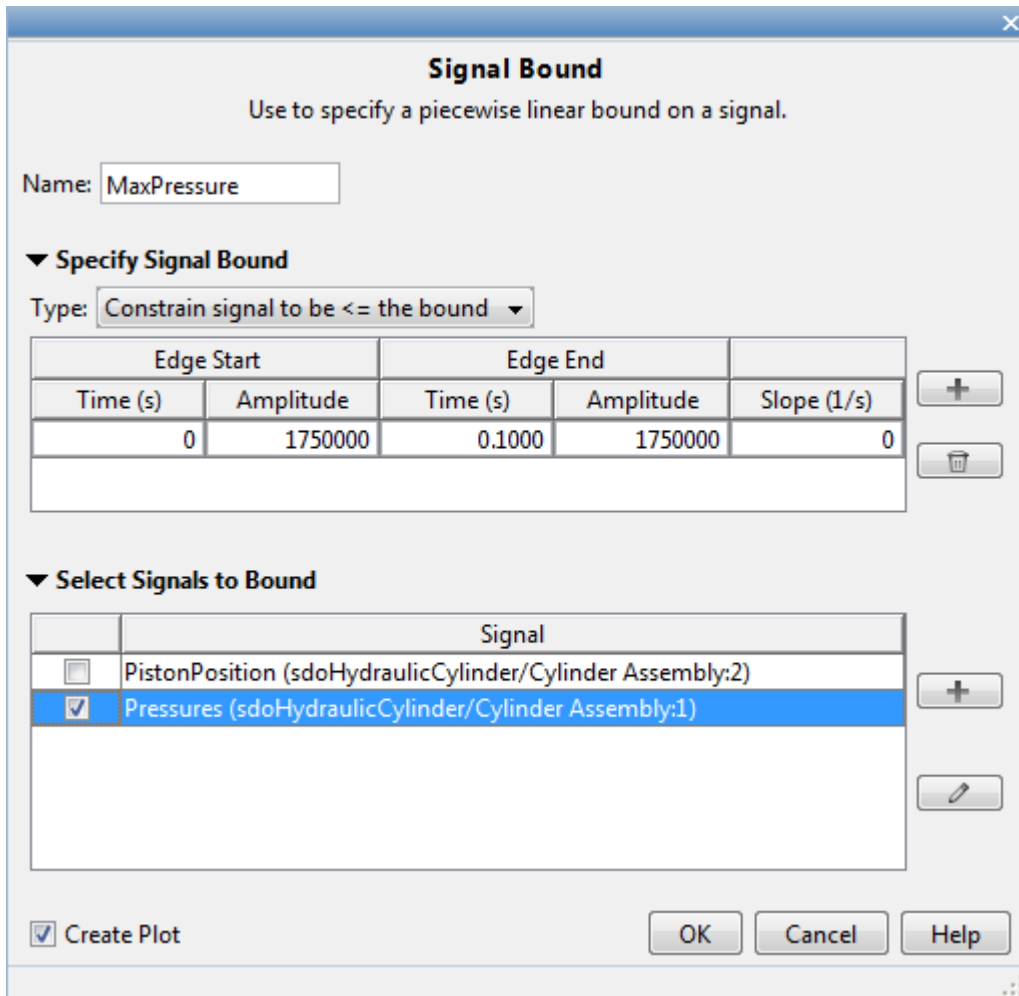
Click **OK**.

- A new `MaxPressure` variable appears in the **Response Optimization Tool Workspace**.

- A graphical view of the maximum pressure requirement is automatically created.
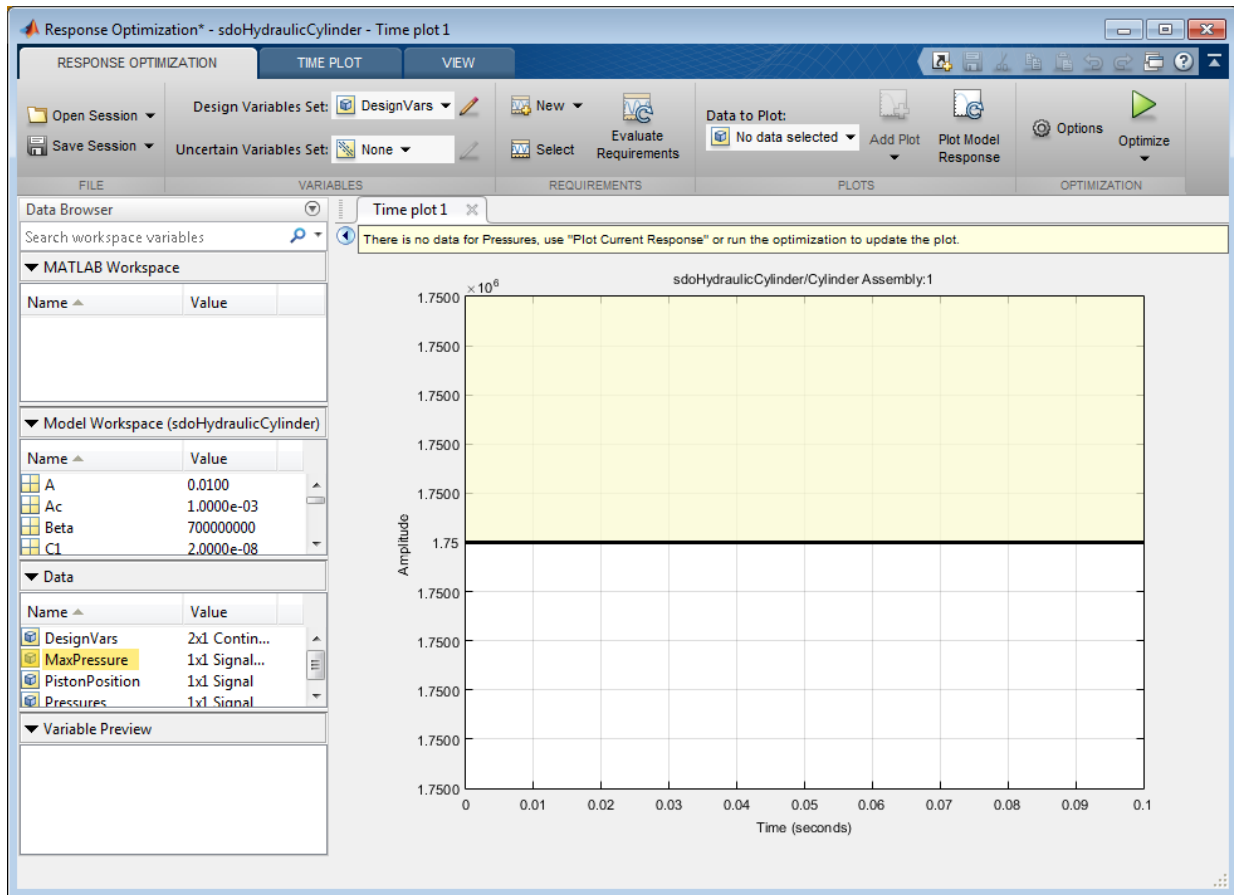
Specify the piston position step response requirement of rise time of less than 0.04 seconds and a settling time of less than 0.05 seconds.

In the **New** drop-down list of the **Response Optimization** tab, select **Step Response Envelope**. A dialog to create a step response requirement opens.

Specify a requirement named `PistonResponse`, and the required rise and settling time bounds. Select `PistonPosition` as the signal to apply the step response requirement to.

Click **OK**.

Specify Custom Objective

The custom objective is to minimize the cylinder cross-sectional area.

In the **New** drop-down list, select **Custom Requirement**. A dialog to create custom requirement opens.
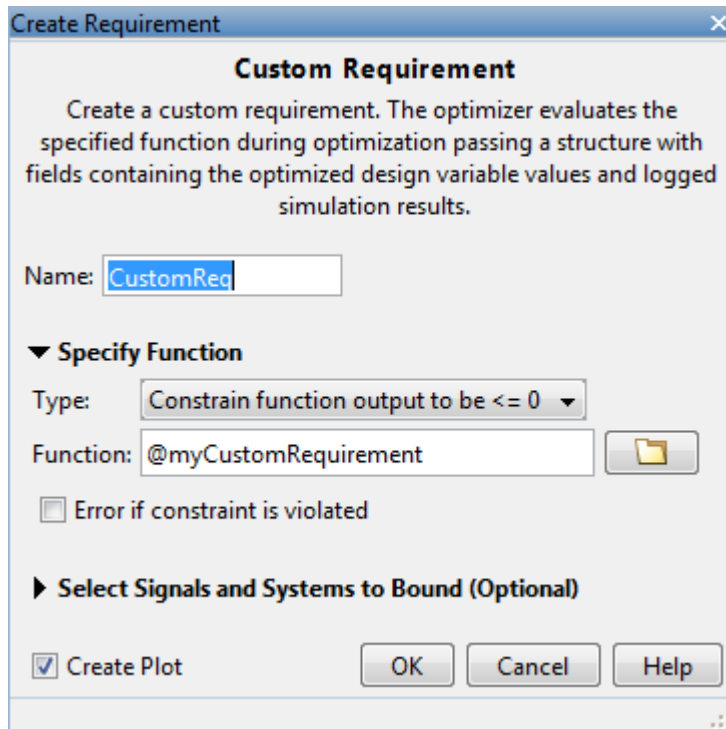
Specify a function to call during optimization in the **Requirement Function** field. At each optimization iteration, the software calls the function and passes the current design variable values. You can also optionally pass logged signals to the custom requirement. Here, you use sdoHydraulicCylinder_customObjective as the custom requirement function, which returns the value of the cylinder cross-sectional area.

In the **Requirement Type** drop-down list, specify whether the requirement is an objective to minimize (min), an inequality constraint (<=), or an equality constraint (==).
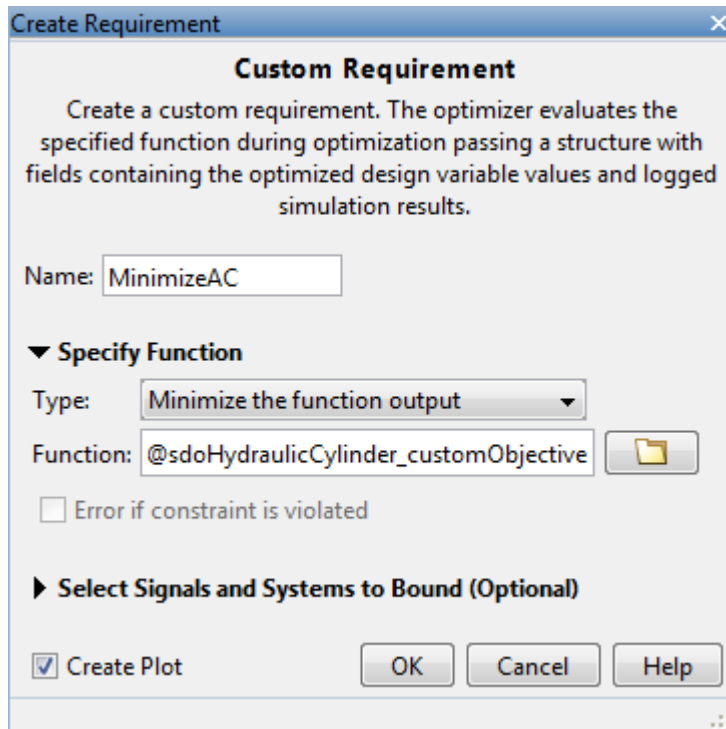
```
type sdoHydraulicCylinder_customObjective
```

```
function objective = sdoHydraulicCylinder_customObjective(data)
%SDOHYDRAULICCYLINDER_CUSTOMOBJECTIVE
%
% The sdoHydraulicCylinder_customObjective function is used to define a
% custom requirement that can be used in the graphical SDTOOL environment.
%
% The |data| input argument is a structure with fields containing the
% design variable values chosen by the optimizer.
%
% The |objective| return argument is the objective value to be minimized by
% the SDOTOOL optimization solver.
%
% Copyright 2011 The MathWorks, Inc.
```

```
%For the cylinder design problem we want to minimize the cylinder
%cross-sectional area so return the cylinder cross-sectional area as an
%objective value.
Ac = data.DesignVars(1);
objective = Ac.Value;
end
```

Evaluate the Initial Design

Click **Plot Model Response** to simulate the model and check how well the initial design satisfies the design requirements. To show both requirement plots at the same time, use the plot layout widgets in the **View** tab.
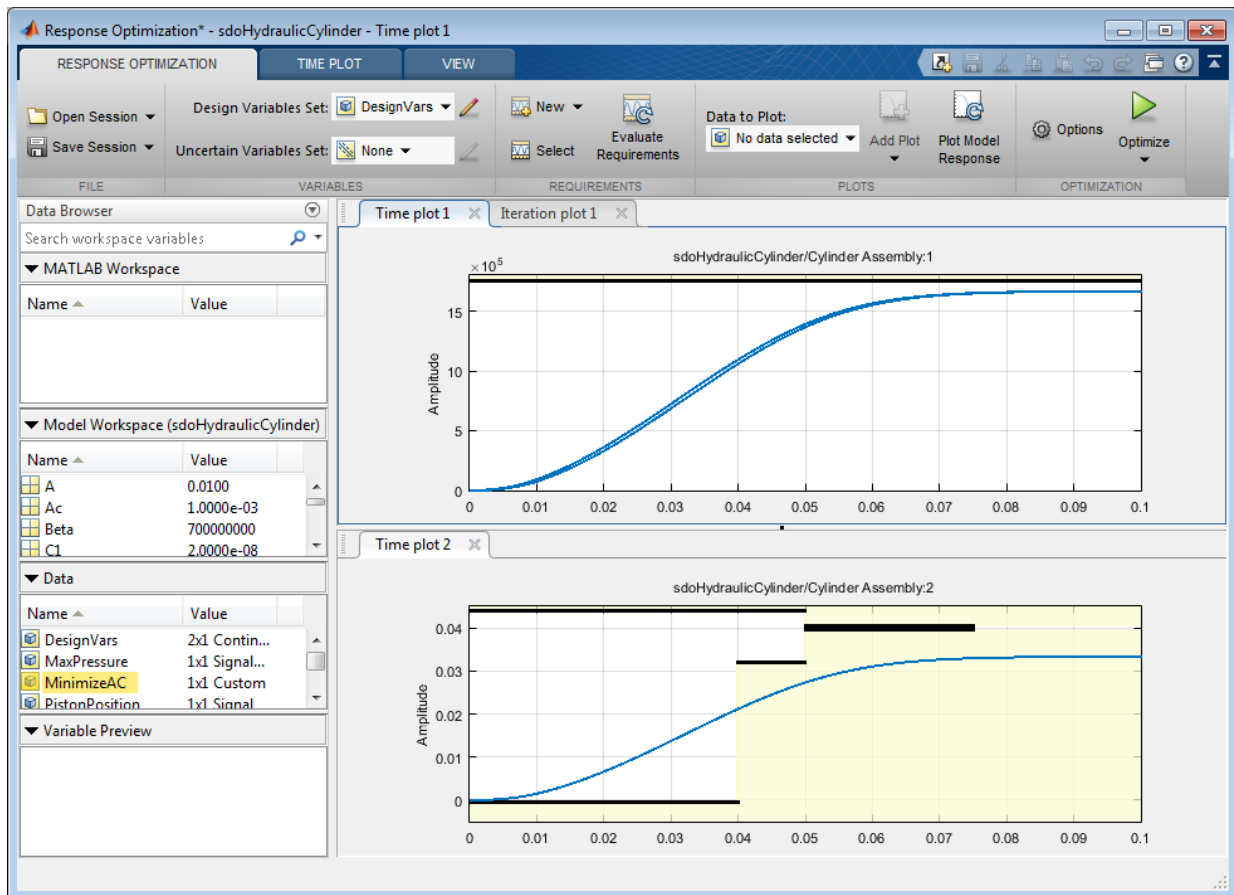
From the plots, see that the maximum pressure requirement is satisfied but the piston position step response requirement is not satisfied.

Optimize the Design

Create a plot to display how the cylinder cross-sectional area and piston spring constant are modified during optimization.

In the **Data to Plot** drop-down list, select **DesignVars**, which contains the optimization design variables Ac and K. In the **Add Plot** drop-down, create a new iteration plot to show the design variable trajectories. For this new plot, click **Show scaled values** in the **Iteration Plot** tab, to facilitate viewing the two trajectories on the same axes.

Click **Optimize** in the **Response Optimization** tab.

Iteration	F-count	MinimizeAC (Minimize)	MaxPressure (<=0)	PistonResponse (<=0)
0	5	1.0000e-03	-0.0480	0.3033
1	11	5.7281e-04	-0.0476	0.0729
2	17	3.8184e-04	-0.0476	-4.2157e-04
3	22	3.7527e-04	-0.0476	-4.7753e-04
4	27	3.7075e-04	-0.0476	-4.0574e-04

Optimization started 19-Oct-2014 19:58:06

Optimization converged, 19-Oct-2014 19:58:13
The optimizer encountered 1 errors during the optimization. Details of the errors have been written to 'EvalErrors' in the Design Optimization workspace.

Save Iteration...   Display Options...   Optimize

The optimization progress window updates at each iteration and shows that the optimization converged after 4 iterations.

The `Pressures` and `PistonPosition` plots indicate that the design requirements are satisfied. The `MinimizeAC` plot shows that the cylinder cross-sectional area `Ac` is minimized.

To view the optimized design variable values, click the variable name in the **Response Optimization Tool Workspace**. The optimized values of the design variables are automatically updated in the Simulink model.

Related Examples

To learn how to optimize the cylinder design using the `sdo.optimize` command, see "Design Optimization to Meet Custom Objective (Code)".

```
% Close the model
delete(sdotool('sdoHydraulicCylinder'))
bdclose('sdoHydraulicCylinder')
```

Design Optimization to Meet a Custom Objective (Code)

This example shows how to optimize a design to meet custom objective using `sdo.optimize`. You optimize the cylinder parameters to minimize the cylinder geometry and satisfy design requirements.

Hydraulic Cylinder Model

Open the Simulink model.

```
sys = 'sdoHydraulicCylinder';
open_system(sys);
```


The hydraulic cylinder model is based on the Simulink model `sldemo_hydcyl`. The model includes:

- `Pump` and `Cylinder Assembly` subsystems. For more information on the subsystems, see "Single Hydraulic Cylinder Simulation".

- A step change applied to the cylinder control valve orifice area that causes the cylinder piston position to change.

Hydraulic Cylinder Design Problem

You tune the cylinder cross-sectional area and piston spring constant to meet the following design requirements:

- Ensure that the piston position has a step response rise time of less than 0.04 seconds and setting time of less than 0.05 seconds.

- Limit the maximum cylinder pressures to 1.75e6 N/m.

- Minimize the cylinder cross-sectional area.

Specify Design Variables

Select the following model parameters as design variables for optimization:

- Cylinder cross-sectional area Ac

- Piston spring constant K

```
Ac = sdo.getParameterFromModel('sdoHydraulicCylinder','Ac');
K  = sdo.getParameterFromModel('sdoHydraulicCylinder','K');
```

Limit the cylinder cross-sectional area to a circular area with radius between 1 and 2 centimeters.

```
Ac.Minimum = pi*1e-2^2; % m^2
Ac.Maximum = pi*2e-2^2; % m^2
```

Limit the piston spring constant to a range of 1e4 to 10e4 N/m.

```
K.Minimum  = 1e4;       % N/m
K.Maximum  = 10e4;      % N/m
```

Specify Design Requirements

The design requirements require logged model signals. During optimization, the model is simulated using the current value of the design variables and the logged signal is used to evaluate the design requirements.

Log the following signals:

- Cylinder pressures, available at the first output port of the Cylinder Assembly block

```
Pressures = Simulink.SimulationData.SignalLoggingInfo;
Pressures.BlockPath       = 'sdoHydraulicCylinder/Cylinder Assembly';
Pressures.OutputPortIndex = 1;
```

- Piston position, available at the second output port of the `Cylinder Assembly` block

```
PistonPosition = Simulink.SimulationData.SignalLoggingInfo;
PistonPosition.BlockPath       = 'sdoHydraulicCylinder/Cylinder Assembly';
PistonPosition.OutputPortIndex = 2;
```

Create an object to store the logging information and use later to simulate the model

```
simulator = sdo.SimulationTest('sdoHydraulicCylinder');
simulator.LoggingInfo.Signals = [PistonPosition,Pressures];
```

Specify the piston position step response requirement of rise time of less than 0.04 seconds and settling time less than of 0.05 seconds.

```
PistonResponse = sdo.requirements.StepResponseEnvelope;
set(PistonResponse, ...
    'RiseTime',        0.04, ...
    'FinalValue',      0.04, ...
    'SettlingTime',    0.05, ...
    'PercentSettling', 1);
```

Specify the maximum cylinder pressure requirement of less than 1.75e6 N/m.

```
MaxPressure = sdo.requirements.SignalBound;
set(MaxPressure, ...
    'BoundTimes',      [0 0.1], ...
    'BoundMagnitudes', [1.75e6 1.75e6], ...
    'Type',            '<=');
```

For convenience, collect the performance requirements into a single structure to use later.

```
requirements = struct(...
    'PistonResponse', PistonResponse, ...
    'MaxPressure',    MaxPressure);
```

Create Objective/Constraint Function

To optimize the cylinder cross-sectional area and piston spring constant, create a function to evaluate the cylinder design. This function is called at each optimization iteration.

Here, use an anonymous function with one argument that calls the
sdoHydraulicCylinder_design function.

```
evalDesign = @(p) sdoHydraulicCylinder_design(p,simulator,requirements);
```

The function:

- Has one input argument that specifies the cylinder cross-sectional area and piston spring constant values.

- Returns the optimization objective value and optimization constraint violation values.

The optimization solver minimizes the objective value and attempts to keep the optimization constraint violation values negative. Type help sdoExampleCostFunction for more details on how to write the objective/constraint function.

The sdoHydraulicCylinder_design function uses the simulator and requirements objects to evaluate the design. Type edit sdoHydraulicCylinder_design to examine the function in more detail.

type sdoHydraulicCylinder_design

```
function design = sdoHydraulicCylinder_design(p,simulator,requirements)
%SDOHYDRAULICCYLINDER_DESIGN
%
% The sdoHydraulicCylinder_design function is used to evaluate a cylinder
% design.
%
% The |p| input argument is the vector of cylinder design parameters.
%
% The |simulator| input argument is a sdo.SimulinkTest object used to
% simulate the |sdoHydraulicCylinder| model and log simulation signals
%
% The |requirements| input argument contains the design requirements used
% to evaluate the cylinder design
%
% The |design| return argument contains information about the design
% evaluation that can be used by the |sdo.optimize| function to optimize
% the design.
%
% see also sdo.optimize, sdoExampleCostFunction

% Copyright 2011 The MathWorks, Inc.
```

```
%% Simulate the model
%
% Use the simulator input argument to simulate the model and log model
% signals.
%
% First ensure that we simulate the model with the parameter values chosen
% by the optimizer.
%
simulator.Parameters = p;
% Simulate the model and log signals.
%
simulator = sim(simulator);
% Get the simulation signal log, the simulation log name is defined by the
% model |SignalLoggingName| property
%
logName = get_param('sdoHydraulicCylinder','SignalLoggingName');
simLog = get(simulator.LoggedData,logName);

%% Evaluate the design requirements
%
% Use the requirements input argument to evaluate the design requirements
%
% Check the PistonPosition signal against the stepresponse requirement
%
PistonPosition = get(simLog,'PistonPosition');
cPiston = evalRequirement(requirements.PistonResponse,PistonPosition.Values);
% Check the Pressure signals against the maximum requirement
%
Pressures = find(simLog,'Pressures');
cPressure = evalRequirement(requirements.MaxPressure,Pressures.Values);
% Use the PistonResponse and MaxPressure requirements as non-linear
% constraints for optimization.
design.Cleq = [cPiston(:);cPressure(:)];
% Add design objective to minimize the Cylinder cross-sectional area
Ac = p(1);           %Since we called sdo.optimize(evalDesign,[Ac;K])
design.F = Ac.Value;
end
```

Evaluate the Initial Design

Call the objective function with the initial cylinder cross-sectional area and initial piston spring constant.

```
initDesign = evalDesign([Ac;K]);
```

The function simulates the model and evaluates the design requirements. The scope shows that the maximum pressure requirement is satisfied but the piston position step response requirement is not satisfied.

`initDesign` is a structure with the following fields:

- `Cleq` shows that some of the inequality constraints are positive indicating they are not satisfied by the initial design.

```
initDesign.Cleq
```

```
ans =

  -0.3839
  -0.1861
  -0.1836
  -1.0000
   0.3033
   0.2909
   0.1671
   0.2326
  -0.0480
  -0.0480
```

- `F` shows the optimization objective value (in this case the cylinder cross-sectional area). The initial design cross-sectional area, as expected, has the same value as the initial cross-sectional area parameter `Ac`.

```
initDesign.F
```

```
ans =

   1.0000e-03
```

Optimize the Design

Pass the objective function, initial cross-sectional area and piston spring constant values to `sdo.optimize`.

```
[pOpt,optInfo] = sdo.optimize(evalDesign,[Ac;K]);
```

```
 Optimization started 20-Jan-2015 08:49:17

                                max        Step-size    First-order
 Iter F-count          f(x)    constraint               optimality
    0     5          0.001       0.3033
    1    11     0.00057281      0.07293        0.48          85.4
    2    17    0.000391755            0       0.128            28
    3    22    0.000388463            0     0.00232       0.00409
    4    27    0.000382784            0     0.00401       0.00231
    5    32    0.000378554            0     0.00299      0.000545
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints are satisfied to within the selected value of the constraint tolerance.
```

The optimization repeatedly evaluates the cylinder design by adjusting the cross-sectional area and piston spring constant to meet the design requirements. From the scope, see that the maximum pressure and piston response requirements are met.

The `sdo.optimize` function returns:

• `pOpt` shows the optimized cross-sectional area and piston spring constant values.

`pOpt`

```
pOpt(1,1) =

       Name: 'Ac'
      Value: 3.7855e-04
    Minimum: 3.1416e-04
    Maximum: 0.0013
       Free: 1
      Scale: 0.0020
       Info: [1x1 struct]


pOpt(2,1) =

       Name: 'K'
      Value: 1.5816e+04
    Minimum: 10000
    Maximum: 100000
```

```
        Free: 1
       Scale: 65536
        Info: [1x1 struct]


2x1 param.Continuous
```

- `optInfo` is a structure that contains optimization termination information such as number of optimization iterations and the optimized design.

`optInfo`

```
optInfo =

            Cleq: [10x1 double]
               F: 3.7855e-04
       Gradients: [1x1 struct]
        exitflag: 1
      iterations: 5
     SolverOutput: [1x1 struct]
           Stats: [1x1 struct]
```

For example, the `Cleq` field shows the optimized non-linear inequality constraints are all non-positive to with-in optimization tolerances, indicating that the maximum pressure and piston response requirements are satisfied.

`optInfo.Cleq`

```
ans =

   -0.0968
   -0.0126
   -0.0126
   -1.0000
   -0.2067
   -0.0052
   -0.0074
   -0.0004
   -0.0476
   -0.0476
```

The F field contains the optimized cross-sectional area. The optimized cross-sectional area value is nearly 50% less that the initial value.

```
optInfo.F
```

```
ans =

   3.7855e-04
```

Update the Model Variable Values

By default, the model variables Ac and K are not updated at the end of optimization. Use the setValueInModel command to update the model variable values.

```
sdo.setValueInModel('sdoHydraulicCylinder',pOpt)
```

Related Examples

To learn how to optimize the cylinder design using the Response Optimization tool, see "Design Optimization to Meet Custom Objective (GUI)".

```
% Close the model
bdclose('sdoHydraulicCylinder')
```

Design Optimization to Meet Custom Signal Requirements (GUI)

This example shows how to optimize a design to meet a custom signal requirement. You optimize the controller parameters to minimize the plant actuation signal energy while satisfying step response requirements.

1   Load a saved Response Optimization tool session.

```
load sldo_model1_custom_signal_session
sdotool(SDOSessionData);
```

The following Simulink model opens.
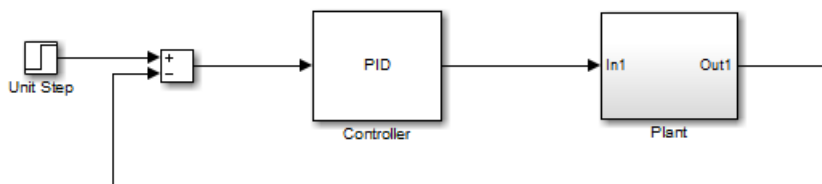
The Response Optimization tool, configured with the following settings, also opens:

- Step response characteristics, specified on the output of the Plant block, that the model output must satisfy:

  - Maximum overshoot of 5%
  - Maximum rise time of 10 seconds
  - Maximum settling time of 30 seconds

- Design variable set with the controller parameters Kp, Ki and Kd. These parameters have a minimum value of 0.

- The variables for step requirements (PlantResponse), logged signal (PlantOutput) and design variables (DesignVars) which appear in the **Data** area.
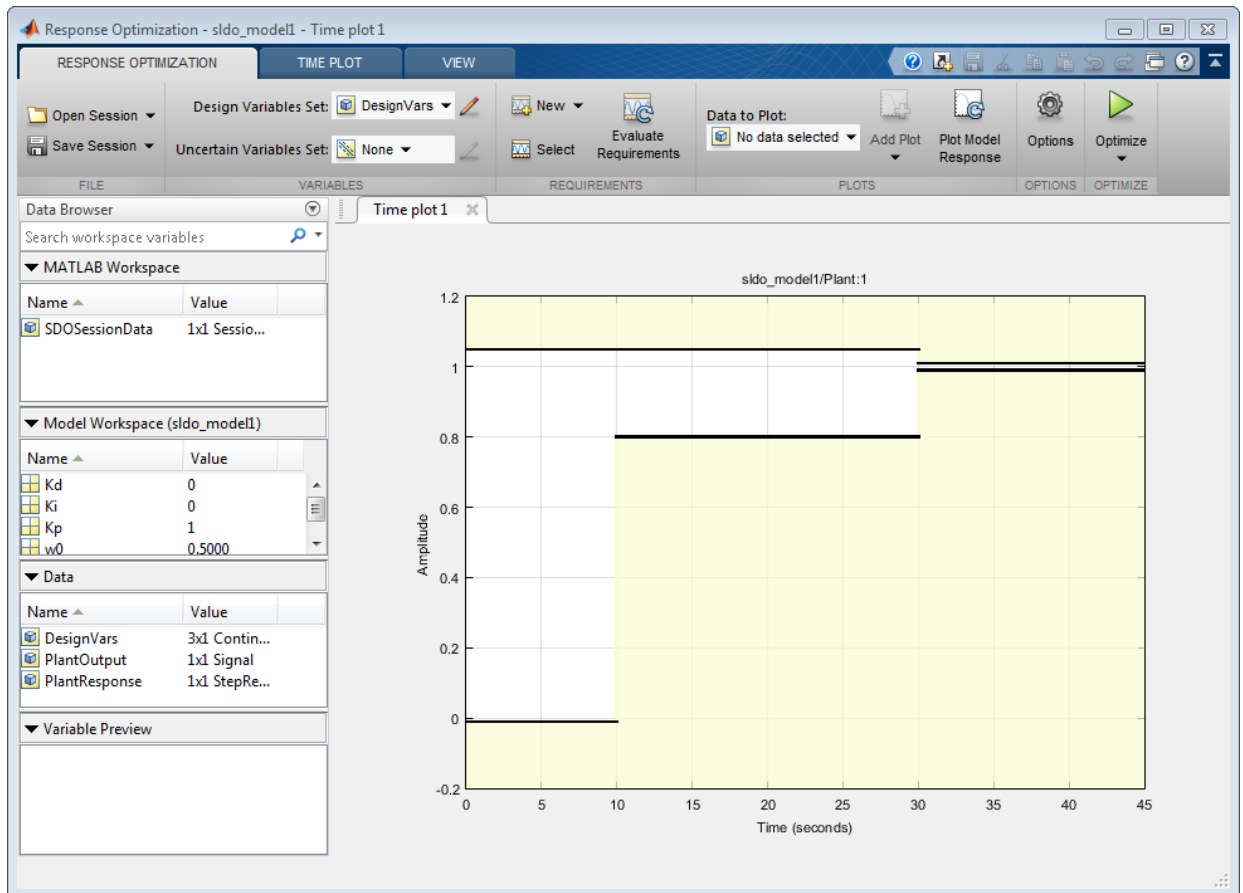
2   Specify a signal to log. You apply the custom requirement on this logged signal.

a   Select **New** > **Signal**.

A window opens where you select a signal to log.

b   In the Simulink model window, click the output of the `Controller` block.

The window updates to display the selected signal.

c
Select the signal and click ⇨ to add it to the signal set.

    d   In **Signal set**, enter `PlantActuator`.

        Click **OK**. A new variable `PlantActuator` appears in the **Data** area.

3   Specify the custom requirement to apply to the signal.

The custom requirement calls the objective function `sldo_model1_minimize_energy` which returns the energy in the `PlantActuator` signal. The signal energy is minimized. This function accepts:

- An input argument `data` which is a structure with fields for the design variables in the **Data** area. Signals are logged for the nominal and uncertain parameter values if there are any.
- Returns the objective value to be minimized.

Tip To see the contents of this function, type `edit sldo_model1_minimize_energy`.

---

    a   Select **New** > **Custom Requirement**.

        A window opens where you specify the custom requirement.

    b   Specify `MinimizeEnergy` as the **Name**.

    c   Specify `@sldo_model1_minimize_energy` as the **Function**.

    d   Select `Minimize the function output` as the **Type**.
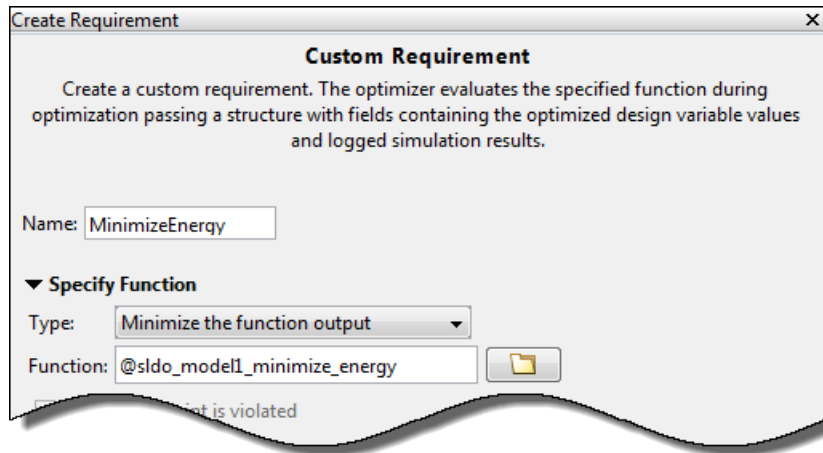
4   In the **Select Signals and Systems to Bound** area, select the `PlantActuator` check box to associate the custom requirement with that signal.
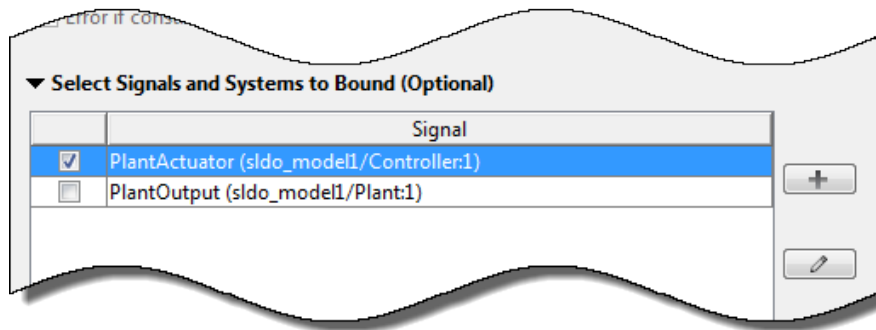
Click **OK**. A new variable appears in the **Data** area of the tool. The window also updates to graphically display the custom signal requirement.

5   Click **Optimize**.
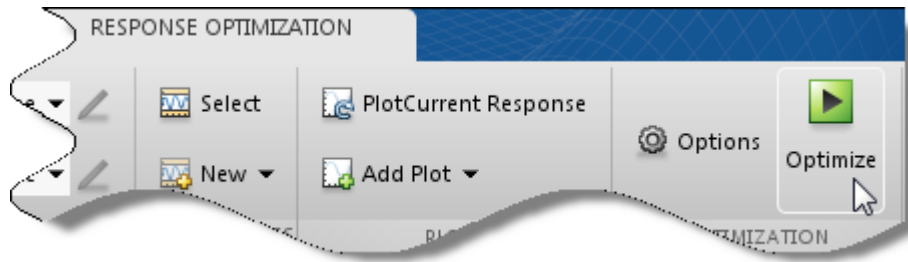
After a few iterations, the optimization converges to meet both the custom signal and step response requirements.

Iteration	F-count	MinimizeEnergy (Minimize)	PlantResponse (<=0)
0	5	328.9064	201.4682
1	12	3.5137e+04	8.5712
2	19	2.4809e+04	1.9011
3	26	3.4691e+04	0.3936
4	33	5.1240e+04	0.4952
5	40	5.0944e+04	0.0613
6	49	2.9535e+04	3.9387e-04
7	58	3.0751e+04	-0.0046
8	71	3.0748e+04	-0.0045
9	79	3.0585e+04	-0.0019
10	100	3.0585e+04	-0.0019

Optimization started 26-Nov-2014 10:46:49

Optimization converged, 26-Nov-2014 10:47:23

Optimized variable values written to 'DesignVars' in the Design Optimization workspace

6 Close the model.

```
setOption(sdotool('sldo_model1'),'NoPromptClose',true)
bdclose('sldo_model1')
```

Related Examples

- "Design Optimization Using Frequency-Domain Check Blocks (GUI)"

Design Optimization to Meet Frequency-Domain Requirements (GUI)

This example shows how to tune model parameters to meet frequency-domain requirements using the Response Optimization tool.

The example requires Simulink® Control Design™

Suspension Model

Open the Simulink Model.

```
open_system('sdoSimpleSuspension')
```

Simple Suspension System
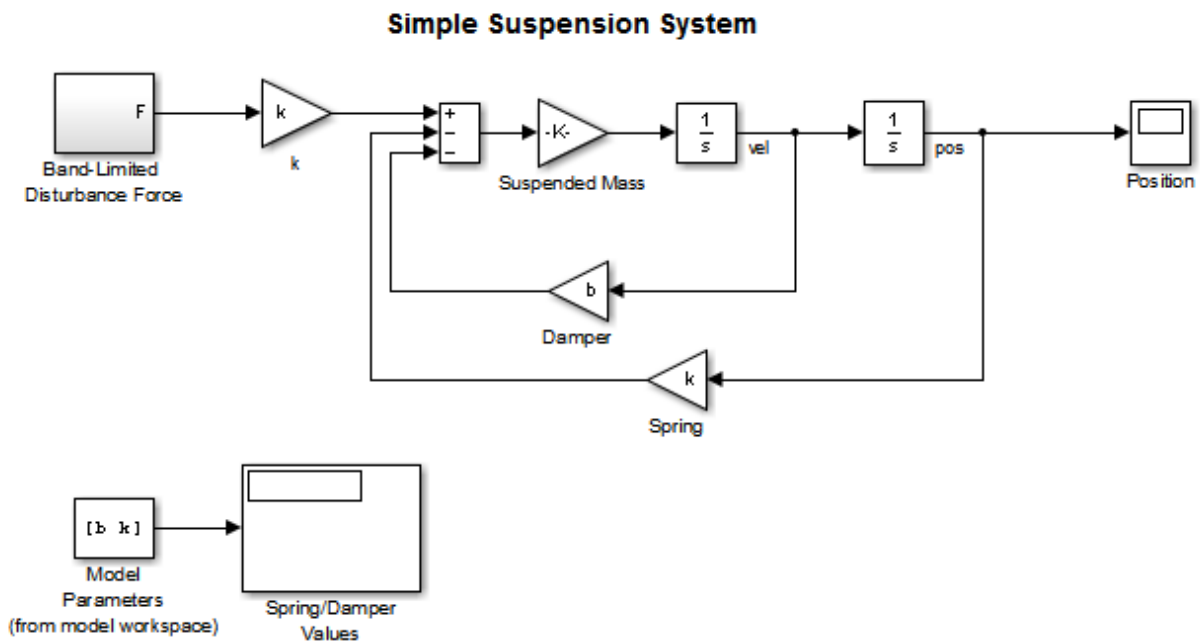
Copyright 2002-2011 The MathWorks, Inc.

Mass-spring-damper models represent simple suspension systems and for this example we tune the system to meet typical suspension requirements. The model implements

the second order system representing a mass-spring-damper using Simulink blocks and includes:

- a `Mass` gain block parameterized by the total suspended mass, `m0+mLoad`. The total mass is the sum of a nominal mass `m0` and a variable load mass `mLoad`.

- a `Damper` gain block parameterized by the damping coefficient, `b`.

- a `Spring` gain block parameterized by the spring constant, `k`.

- two integrator blocks to compute the mass velocity and position.

- a `Band-Limited Disturbance Force` block applying a disturbance force to the Mass. The disturbance force is assumed to be band-limited white noise.

Simulate the model to view the system response to the applied disturbance force.
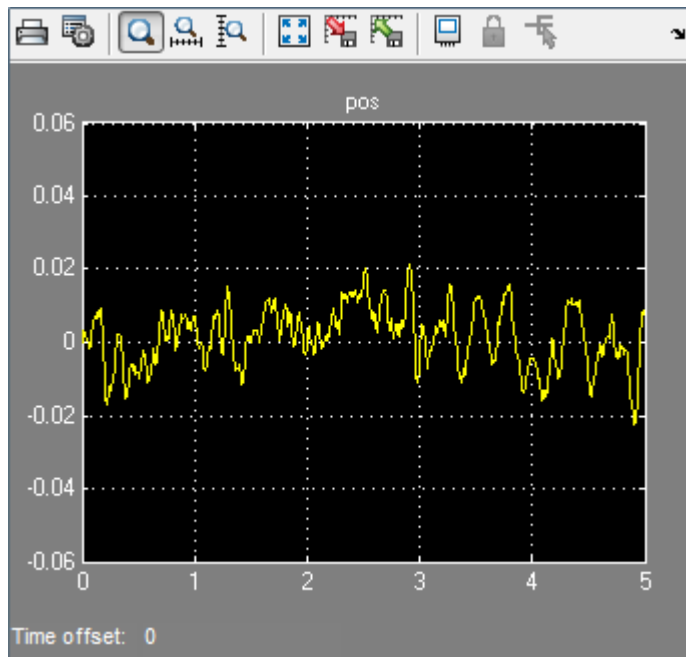
Design Problem

The initial system has a bandwidth that is too high. This can be seen from the spiky position signal. You tune the spring and damper values to meet the following requirements:

- The -3dB system bandwidth must not exceed 10 rad/s.

- The damping ratio of the system must be less than 1/sqrt(2). This ensures that no frequencies in pass band are amplified by the system.

- Minimize the expected failure rate of the system. The expected failure rate is described by a Weibull distribution dependent on the mass, spring, and damper values.

- These requirements must all be satisfied as the load mass ranges from 0 to 20.

Open the Response Optimization Tool

In the Simulink model **Analysis** menu, select **Response Optimization**.

Specify Design Variables

In the **Design Variables Set** list, select **New**. Add the b and k model variables to the design variable set.

- Specify the Minimum and Maximum values for the b variable as 100 and 10000 respectively.

- Specify the Minimum and Maximum values for the k variable as 10000 and 100000 respectively.
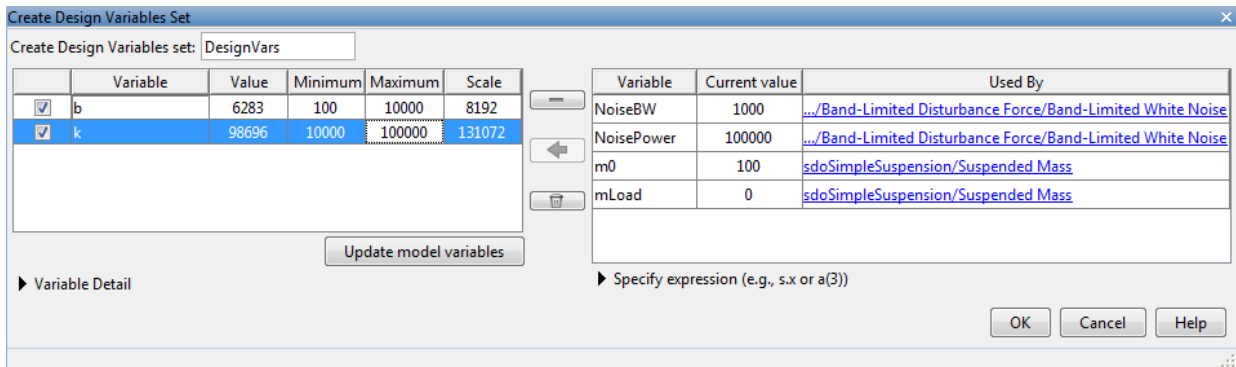
Click **OK**. A new variable, `DesignVars`, appears in the **Response Optimization Tool Workspace**.

In the **Uncertain Variables Set** list, select **New**. Add the `mLoad` variable to the uncertain variables set.

- Specify the `Uncertain Values` value for the `mLoad` variable as [10 15 20]
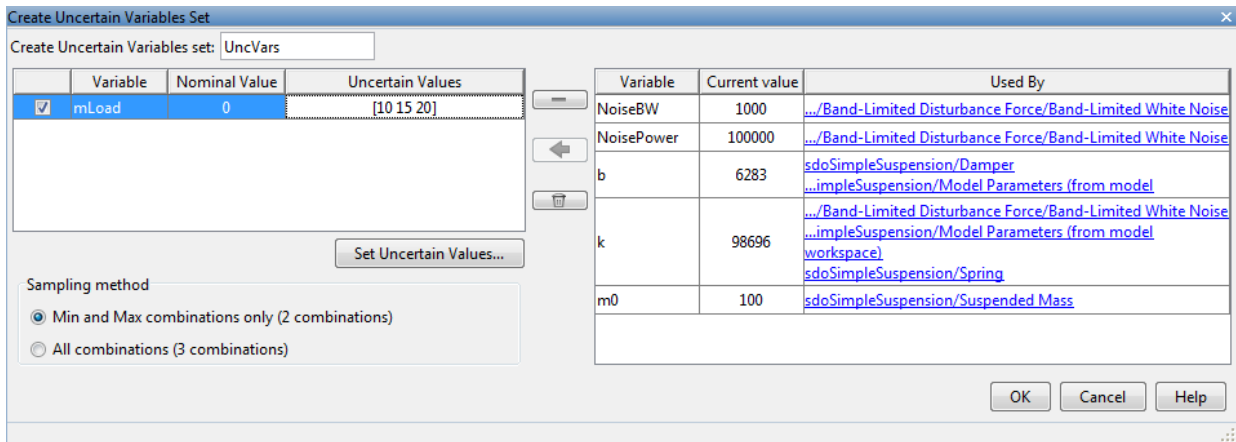
Click **OK**. A new variable, `UncVars`, appears in the **Response Optimization Tool Workspace**.

Specify Linear Analysis Input/Output Points

Specify the input/output points defining the linear system used to compute the bandwidth and damping ratio.

To specify the input/output points:

- In the **New** list, select **Linearization I/Os**.

- In the Simulink model, click the signal at the output of the `Band-Limited Disturbance Force` block. The **Create linearization I/O set** dialog box is updated and the chosen signal appears in it.

- In the **Create linearization I/O** dialog box, select the signal and click **Add**.

- In the **Configuration** list for the selected signal, choose **Input Perturbation** to specify it as an input signal.
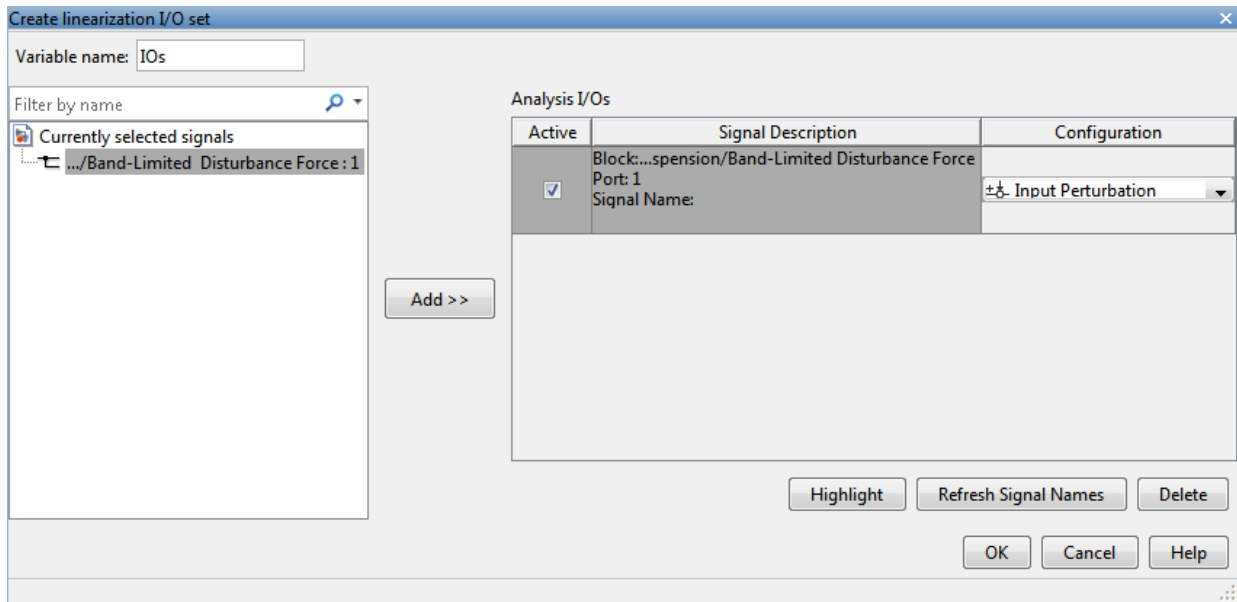
- Similarly, add the **pos** signal from the Simulink model. Specify this signal as an output. In the **Configuration** list, select **Output Measurement**.
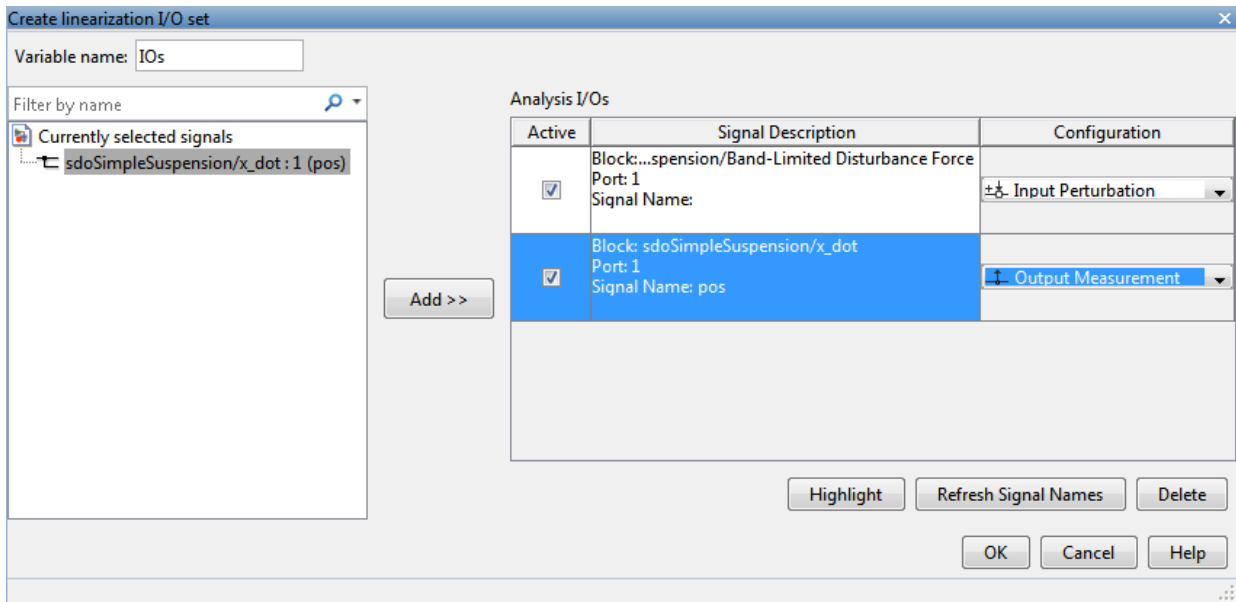
- Click **OK**. A new variable, `IOs`, appears in the **Response Optimization Tool Workspace**.

Add Bandwidth and Damping-Ratio Requirements

Tune the spring and damper values to satisfy bandwidth and damping ratio requirements.

To specify the bandwidth requirement:

- Open a dialog to specify bounds on the Bode magnitude. In the **New** list, select **Bode Magnitude**.

- Specify the requirement name as `Bandwidth`.

- Specify the edge start frequency and magnitude as 10 rad/s and -3db, respectively.

- Specify the edge end frequency and magnitude as 100 rad/s and -3db, respectively.

- Specify the input/output set to which the requirement applies by clicking **Select Systems to Bound**. Select the `IOs` check box .

Create Requirement ✕

Bode Magnitude

Specify a piecewise linear bound on the frequency response of a linear system.

Name: Bandwidth

▼ **Specify Magnitude Bound**

Type: Constrain system to be <= the bound ▼

Edge Start		Edge End		
Freq. (rad/s)	Mag. (dB)	Freq. (rad/s)	Mag. (dB)	Slope (dB/dec...
10.0000	-3	100	-3	0

▼ **Select Systems to Bound**

Snapshot Times: [0]

	Linearization I/O
☑	IOs (sdoSimpleSuspension/Band-Limited Disturbance Force:1 [in], sdoSim...

☑ Create Plot   OK   Cancel   Help
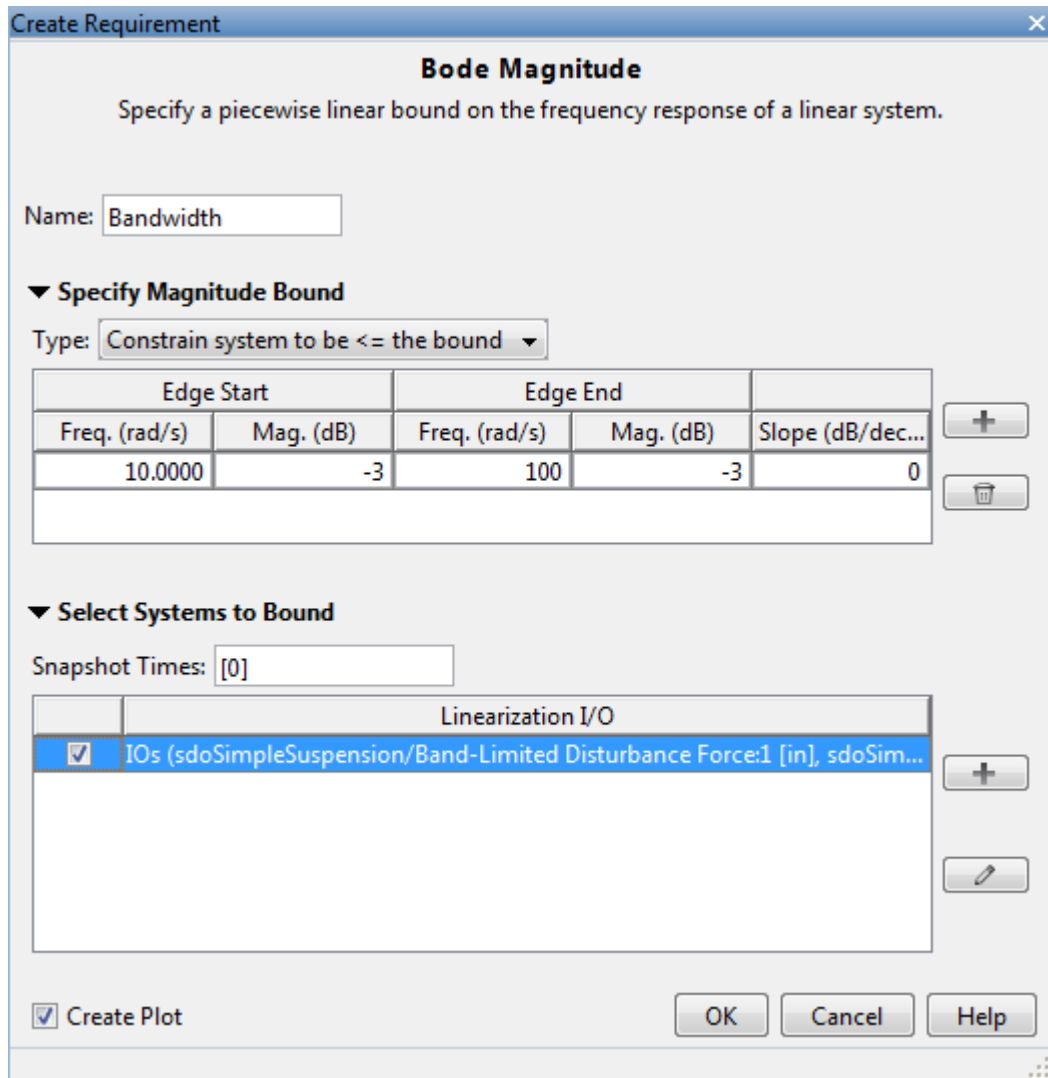
- Click **OK**. A new requirement, Bandwidth, appears in the **Response Optimization Tool Workspace** and a graphical view of the bandwidth requirement is automatically created.

To specify the damping ratio requirement:

- Open a dialog to specify bounds on the damping ratio. In the **New** list, select **Damping Ratio**.

- Specify the damping ratio bound value as 0.7071.

- Specify the input/output set to which the requirement applies by clicking **Select Systems to Bound**. Select the IOs check box .
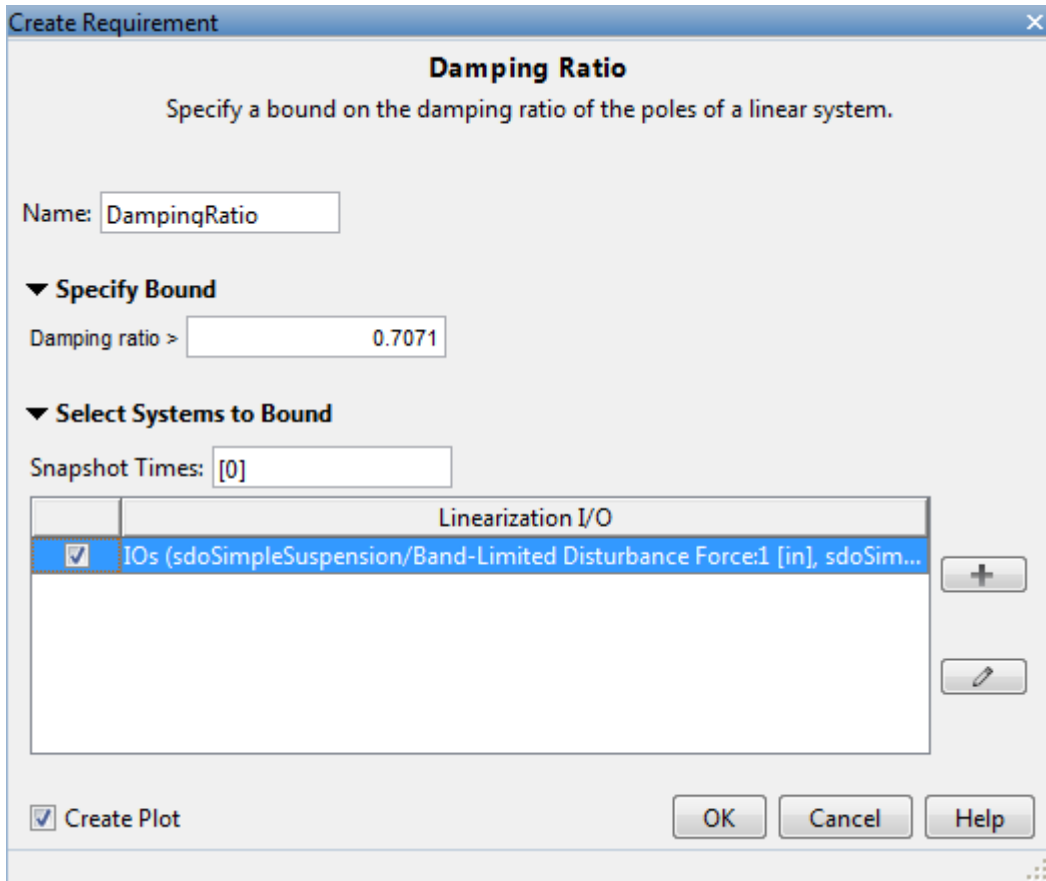
- Click **OK**. A new requirement, DampingRatio, appears in the **Response Optimization Tool Workspace** and a graphical view of the damping ratio requirement is automatically created.

Add a Reliability Requirement

Tune the spring and damper values to minimize the expected failure rate over a lifetime of 100e3 miles. The failure rate is computed using a Weibull distribution on the damping ratio of the system. As the damping ratio increases the failure rate is expected to increase.

Specify the reliability requirement as a custom requirement:

- Open a dialog box to specify the custom requirements. In the **New** list, select **Custom Requirement**.

- Specify the custom requirement name as `MinFailureRate`.

- In the **Specify Function** area, select **Minimize the function output** from the **Type** list.

- Specify the function as @sdoSuspension_FailureRate.
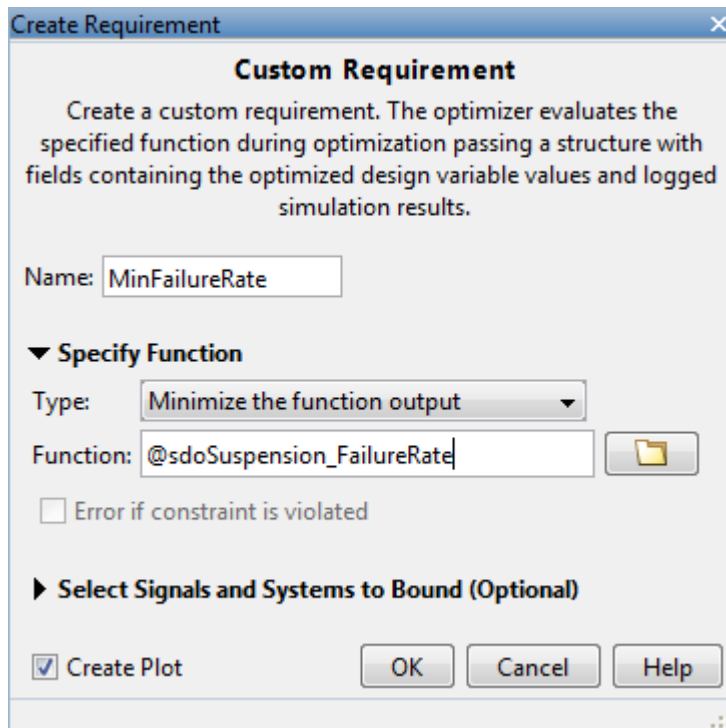
- Click **OK**. A new requirement, `MinFailureRate`, appears in the **Response Optimization Tool Workspace** and a graphical view of the custom requirement is automatically created.

The @sdoSuspension_FailureRate function returns expected failure rate for a lifetime of 100e3 miles.

```
type sdoSuspension_FailureRate
```

```
function pFailure = sdoSuspension_FailureRate(data)
%SDOSUSPENDION_FAILURERATE
%
% The sdoSuspension_FailureRate function is used to define a custom
% requirement that can be used in the graphical SDTOOL environment.
%
% The |data| input argument is a structure with fields containing the
% design variable values chosen by the optimizer.
%
% The |pFailure| return argument is the failure rate to be minimized by the
% SDOTOOL optimization solver. The failure rate is given by a Weibull
% distribution that is a function of the mass, spring and damper values.
% The design minimizes the failure rate for a 100e3 mile lifetime.
%

% Copyright 2012 The MathWorks, Inc.

%Get the spring and damper design values
allVarNames = {data.DesignVars.Name};
idx         = strcmp(allVarNames,'k');
k           = data.DesignVars(idx).Value;
idx         = strcmp(allVarNames,'b');
b           = data.DesignVars(idx).Value;

%Get the nominal mass from the model workspace
wksp = get_param('sdoSimpleSuspension','ModelWorkspace');
m    = evalin(wksp,'m0');

%The expected failure rate is defined by the Weibull cumulative
%distribution function, 1-exp(-(x/l)^k), where k=3, l is a function of the
%mass, spring and damper values, and x the lifetime.
d        = b/2/sqrt(m*k);
pFailure = 1-exp(-(100e3*d/250e3)^3);
end
```

Optimize the Design

Before running the optimization be sure to have completed the earlier steps. Alternatively, you can load the sdoSimpleSuspension_sdosession from the model workspace into the Response Optimization tool.

To save the initial design variable values and later compare them with the optimized values configure the optimization.

• Click **Options**.

• Select the Save optimized variable values as new design variable set option.

To study how the design variable values change during optimization:

• In the **Data to Plot** list, select **DesignVars**.

• In the **Add Plot** list, and select **Iteration Plot**.

• View the design variables in an appropriately scaled manner. Right-click on the DesignVars plot and select **Show scaled values**.

To evaluate the requirements at the initial design point, click **Evaluate Requirements** . The requirement plots are updated and a ReqValues variable is added to the **Response Optimization Tool Workspace**.
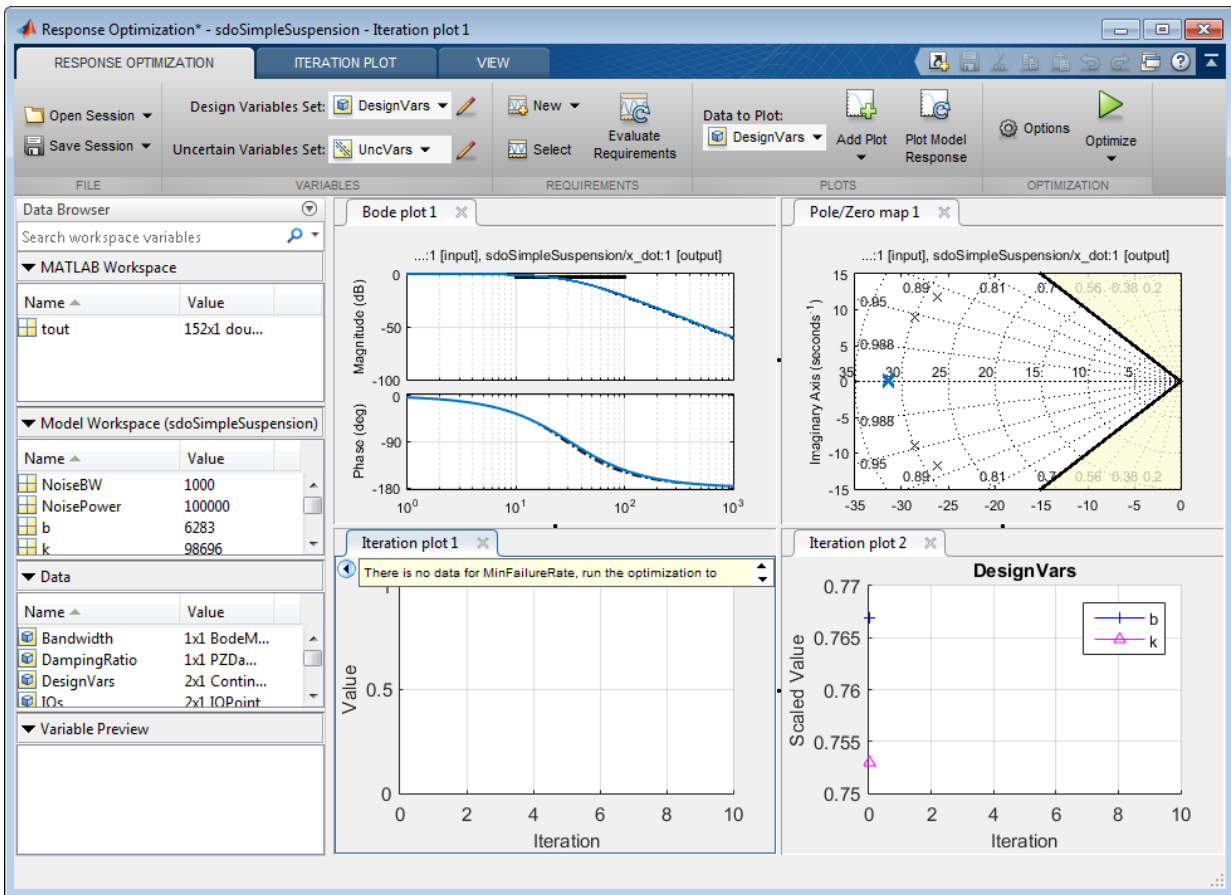
To optimize the design, click **Optimize**. The plots are updated during optimization. At the end of optimization, the optimal design values are written to the `DesignVars1` variable. The requirement values for the optimized design are written to the `ReqValues1` variable.
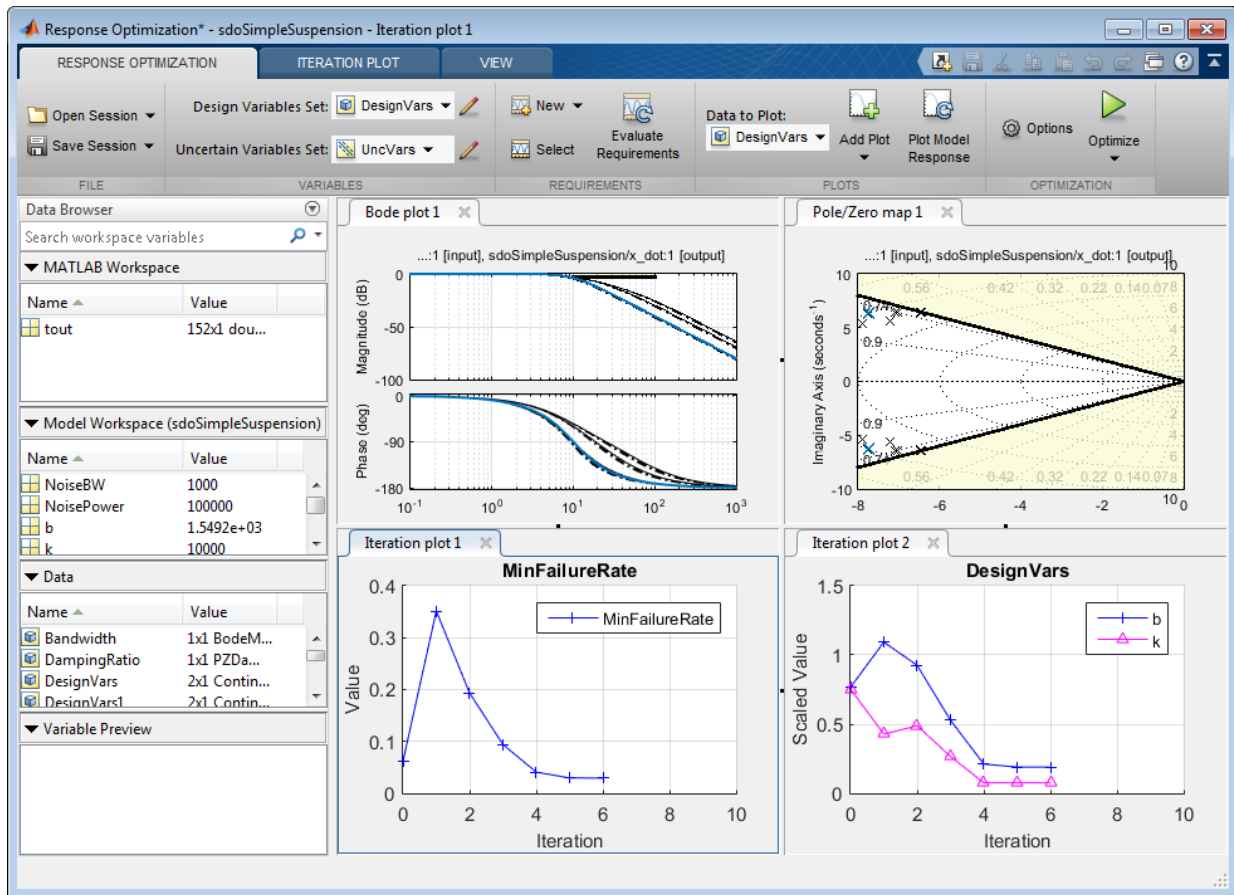
Iteration	F-count	MinFailureRate (min)	Bandwidth (<=0)	DampingRatio (>=0)
0	5	0.0620	0.7228	0.4072
1	10	0.3293	-0.6306	0.4142
2	15	0.1807	-0.0398	0.4142
3	20	0.0801	0.0258	0.4142
4	24	0.0353	-0.4506	0.1612
5	28	0.0269	-0.1833	0.0588
6	32	0.0227	-0.0178	-3.1402e-16
7	37	0.0227	-0.0178	0

Optimization started 26-Mar-2013 16:38:35

Optimization converged, 26-Mar-2013 16:41:02

Optimized variable values written to 'DesignVars1' in the Design Optimization workspace

Save Iteration...   Display Options...   Optimize

Analyze the Design

To compare design variables before and after optimization:

- In the **Data to Plot** list, select **DesignVars**.

- In the **Add Plot** list, select **Spider Plot**.

- To add the optimized design variables to the same plot, select `DesignVars1` in the **Response Optimization Tool Workspace** and drag it onto the Spider plot. Alternatively, in the **Data to Plot** list, select `DesignVars1`. Then, in the **Add Plot** list, select **Spider plot 1** from the **Add to Existing Plot** section.

3-155
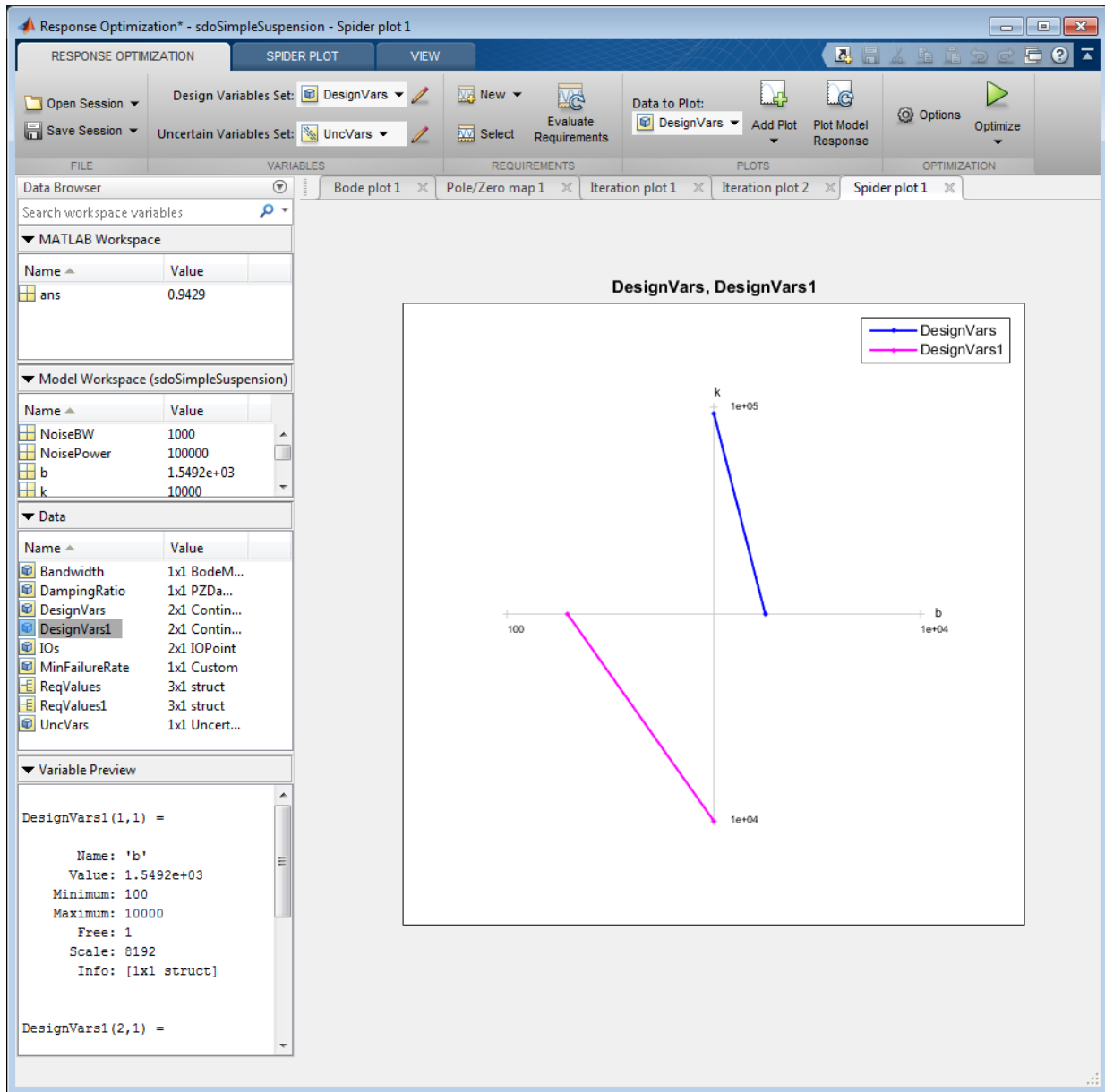
The plot shows that the optimizer reduced both the k and b values for the optimal design.

To compare requirements before and after optimization:

- In the **Data to Plot** list, select **ReqValues**.

- In the **Add Plot** list, select **Spider Plot**.

- To add the optimized requirement values to the same plot, select `ReqValues1` in the **Response Optimization Tool Workspace** and drag it onto the Spider plot. Alternatively, in the **Data to Plot** list, select `ReqValues1`. Then, in the **Add Plot** list, select **Spider plot 2** from the **Add to Existing Plot** section.
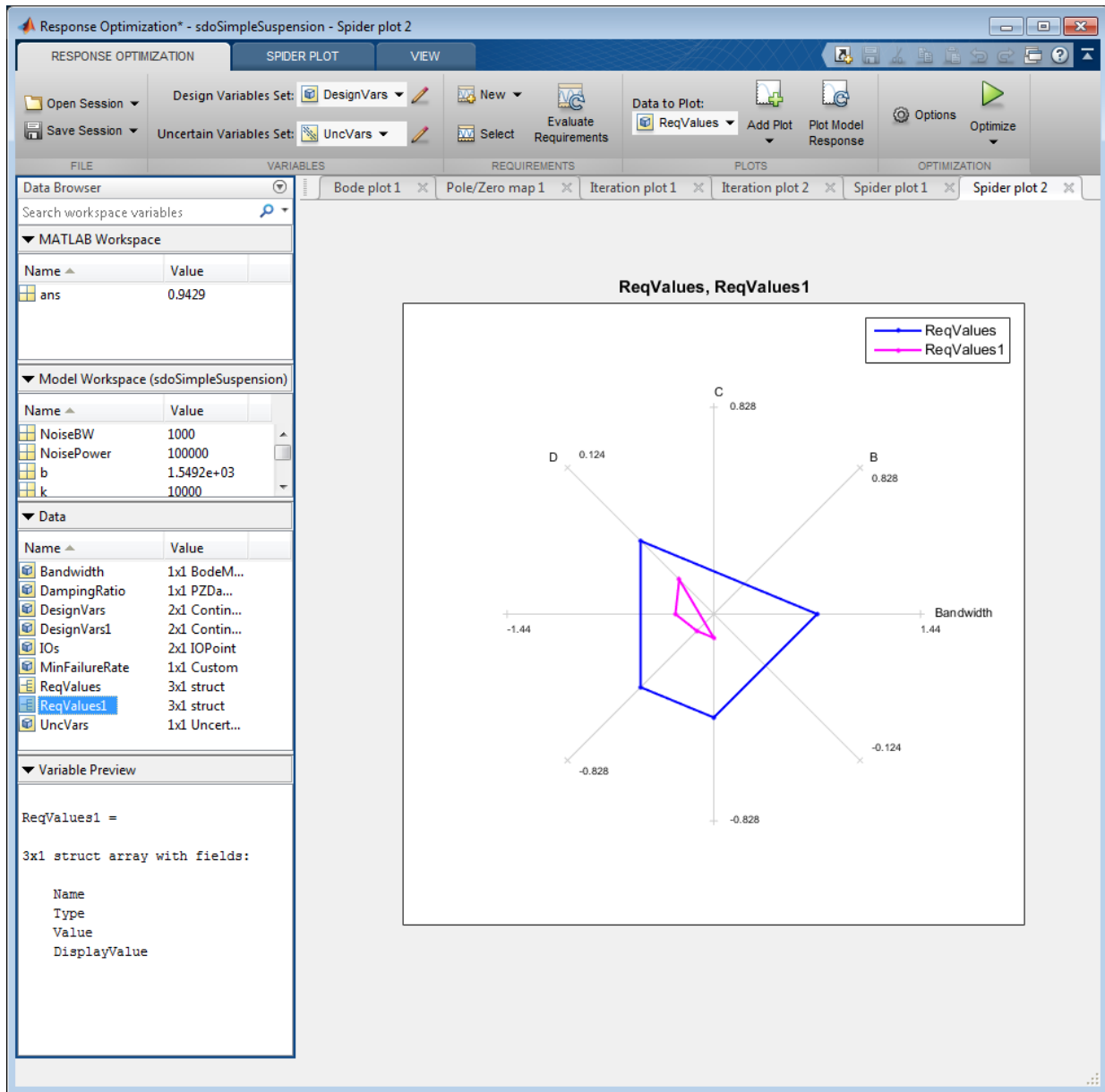
The plot shows that the optimal design has a lower failure rate (the MinFailureRate axis) and better satisfies the bandwidth requirement. The value plotted on the bandwidth axis is the difference between the bandwidth bound and the bandwidth value. The optimization satisfies the bound by keeping this value negative; a more negative value indicates better satisfaction of the bound.

The improved reliability and bandwidth are achieved by pushing the damping ratio closer to the damping ratio bound. The plot has two axes for the damping ratio requirement, one for each system pole, and the plotted values are the difference between the damping ratio bound and the damping ratio value. The optimization satisfies the bound by keeping this value negative.

Finally the simulated mass position is smoother than the initial position response (indication of a lower bandwidth as required) at the expense of larger position deflection.
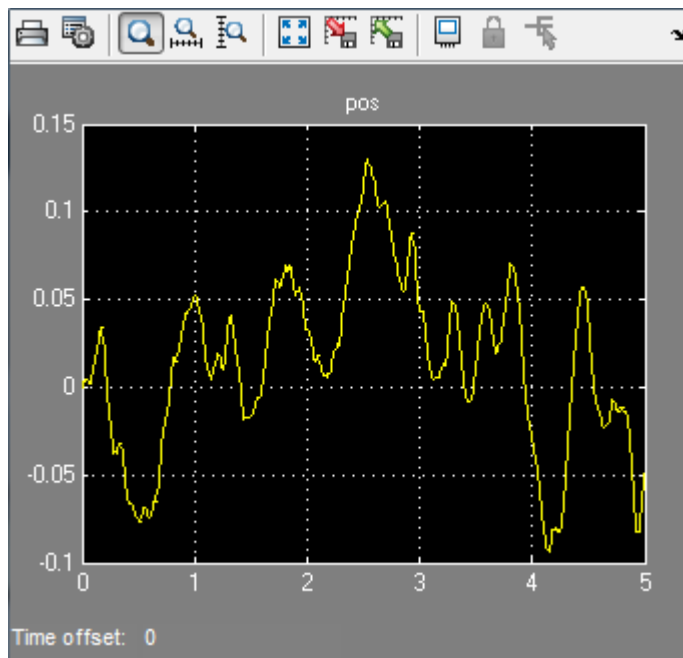

```
% Close the model
bdclose('sdoSimpleSuspension')
```

Specify Custom Signal Objective with Uncertain Variable (GUI)

This example shows how to specify a custom objective function for a model signal. You calculate the objective function value using a variable that models parameter uncertainty.

Competitive Population Dynamics Model

The Simulink model `sdoPopulation` models a simple two-organism ecology using the competitive Lotka-Volterra equations:

$$\frac{dP_1}{dt} = R_1 P_1 (1 - \frac{P_1(t - \tau_1) + \alpha P_2(t - \tau_2)}{K})$$

$$\frac{dP_2}{dt} = R_2 P_2 (1 - \frac{P_2(t - \tau_2) + \alpha P_1(t - \tau_1)}{K})$$

- P_n is the population size of the n-th organism.

- R_n is the inherent per capita growth rate of each organism.

- τ_n is the competitive delay for each organism.

- K is the carrying capacity of the organism environment.

- α is the proximity of the two populations and how strongly they affect each other.

The model uses normalized units.

Open the model.

```
open_system('sdoPopulation')
```
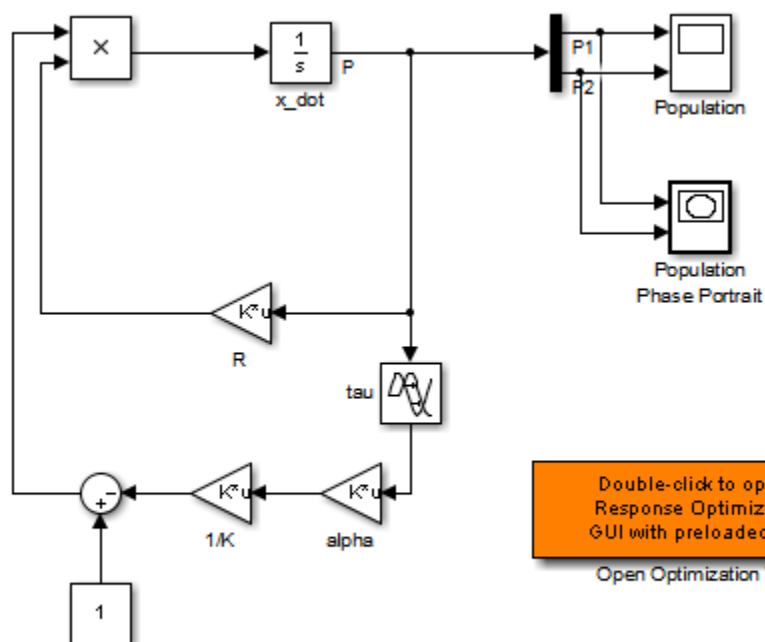
Copyright 2012-2014 The MathWorks, Inc.
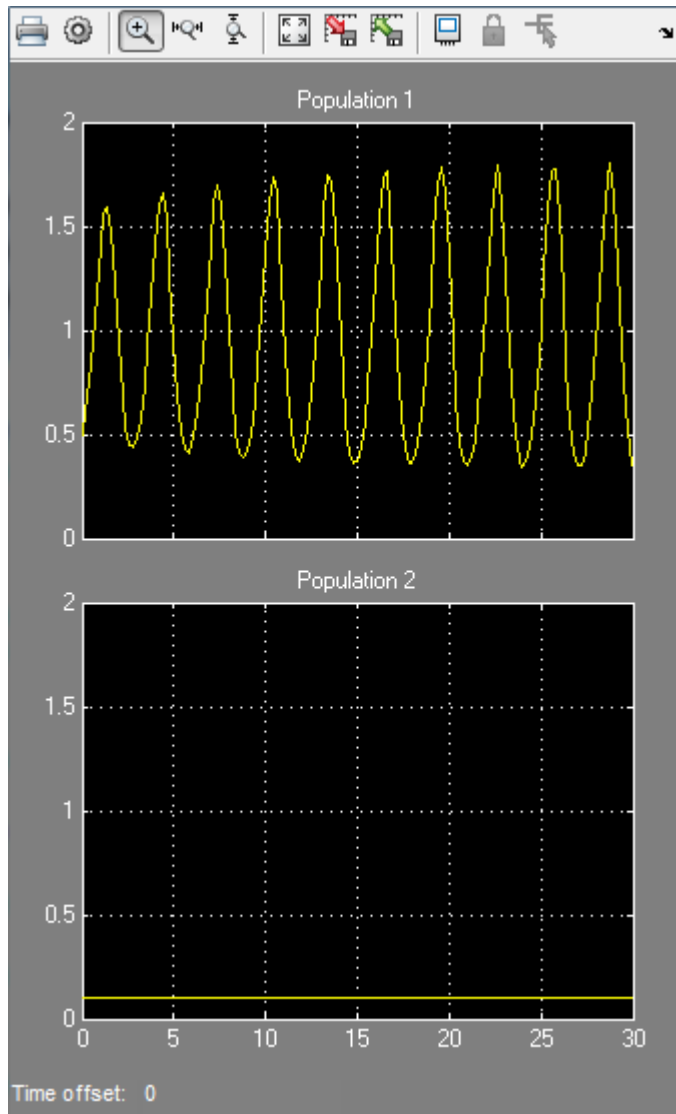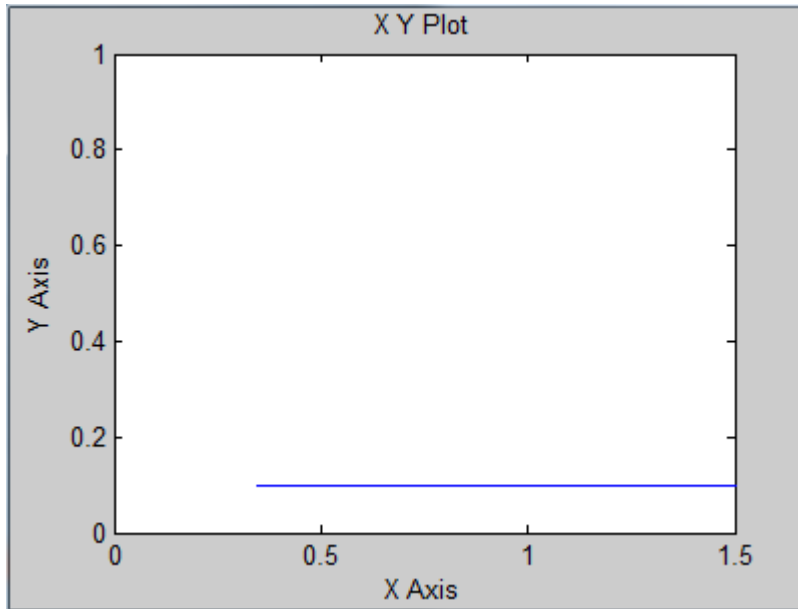
The two-dimensional signal, P, models the population sizes for P1 (first element) and P2 (second element). The model is initially configured with one organism, P1, dominating the ecology. The Population scope shows the P1 population oscillating between high and low values, while P2 is constant at 0.1. The Population Phase Portrait block shows the population sizes of the two organisms in relation to each other.

Population Stabilization Design Problem

Tune the R_2, τ_2, and α values to meet the following design requirements:

- Minimize the population range, that is, the maximum difference between P1 and P2.

- Stabilize P1 and P2, that is, ensure that neither organism population dies off or grows extremely large.

You must tune the parameters for different values of the carrying capacity, K. This ensures robustness to environment carrying-capacity uncertainty.

Open Response Optimization Tool

Double-click the Open Optimization Tool block in the model to open a pre configured Response Optimization tool session. The session specifies the following variables:

- `DesignVars` - Design variables set for the R_2, τ_2, and α model parameters.

- `K_unc` - Uncertain parameter modeling the carrying capacity of the organism environment (K). `K_unc` specifies the nominal value and two sample values.

- `P1` and `P2` - Logged signals representing the populations of the two organisms.

Specify Custom Signal Objective Function

Specify a custom requirement to minimize the maximum difference between the two population sizes. Apply this requirement to the `P1` and `P2` model signals.

1 Open the Create Requirement dialog box. In the **New** list, select **Custom Requirement**.

2 Specify the following in the Create Requirement dialog box:

- **Name** - Enter `PopulationRange`.

- **Type** - Select **Minimize the function output** from the list.

- **Function** - Enter `@sdoPopulation_PopRange`. For more information about this function, see **Custom Signal Objective Function Details**.

- **Select Signals and Systems to Bound (Optional)** - Select the `P1` and `P2` check boxes.

Create Requirement

Custom Requirement

Create a custom requirement. The optimizer evaluates the specified function during optimization passing a structure with fields containing the optimized design variable values and logged simulation results.

Name: PopulationRange

▼ **Specify Function**

Type: Minimize the function output

Function: @sdoPopulation_PopRange

☐ Error if constraint is violated

▼ **Select Signals and Systems to Bound (Optional)**

	Signal
☑	P1 (sdoPopulation/Demux:1)
☑	P2 (sdoPopulation/Demux:2)

Snapshot Times: [0]

	Linearization I/O

Create Linearization I/Os defining a system so that it can be used in requirements.

☑ Create Plot   OK   Cancel   Help

3. Click **OK**.

A new variable, `PopulationRange`, appears in the **Response Optimization Tool Workspace**.

Custom Signal Objective Function Details

`PopulationRange` uses the `sdoPopulation_PopRange` function. This function computes the maximum difference between the populations, across different environment carrying capacity values. By minimizing this value, you can achieve both design goals. The function is called by the optimizer at each iteration step.

To view the function, type `edit sdoPopulation_PopRange`. The following discusses details of this function.

Input/Output

The function accepts `data`, a structure with the following fields:

- `DesignVars` - Current iteration values of R_2, τ_2, and α.

- `Nominal` - Logged signal data, obtained by simulating the model using parameter values specified by `data.DesignVars` and nominal values for all other parameters. The `Nominal` field is itself a structure with fields for each logged signal. The field names are the logged signal names. The custom requirement uses the logged signals, `P1` and `P2`. Therefore, `data.Nominal.P1` and `data.Nominal.P2` are timeseries objects corresponding to `P1` and `P2`.

- `Uncertain` - Logged signal data, obtained by simulating the model using the sample values of the uncertain variable `K_unc`. The `Uncertain` field is a vector of `N` structures, where `N` is the number of sample values specified for `K_unc`. Each element of this vector is similar to `data.Nominal` and contains simulation results obtained from a corresponding sample value specified for `K_unc`.

The function returns the maximum difference between the population sizes across different carrying capacities. The following code snippet in the function performs this action:

```
val = max(maxP(1)-minP(2),maxP(2)-minP(1));
```

Data Time Range

When computing the design goals, discard the initial population growth data to eliminate biases from the initial-condition. The following code snippet in the function performs this action:

```
%Get the population data
tMin = 5; %Ignore signal values prior to this time
iTime = data.Nominal.P1.Time > tMin;
sigData = [data.Nominal.P1.Data(iTime), data.Nominal.P2.Data(iTime)];
```

`iTime` represents the time interval of interest, and the columns of `sigData` contain `P1` and `P2` data for this interval.

Optimization for Different Values of Carrying Capacity

The function includes the effects of varying the carrying capacity by iterating through the elements of `data.Uncertain`. The following code snippet in the function performs this action:

```
...
for ct=1:numel(data.Uncertain)
   iTime = data.Uncertain(ct).P1.Time > tMin;
   sigData = [data.Uncertain(ct).P1.Data(iTime), data.Uncertain(ct).P2.Data(iTime)];

   maxP = max([maxP; max(sigData)]);  %Update maximum if new signals are bigger
   minP = min([minP; min(sigData)]);  %Update minimum if new signals are smaller
end
...
```

The maximum and minimal populations are obtained across all the simulations contained in `data.Uncertain`.

Optimize Design

Click **Optimize**.

The optimization converges after a number of iterations.

Iteration	F-count	PopulationRange (min)
0	8	1.5024
1	13	1.3350
2	20	0.5918
3	29	0.4814
4	36	0.4443
5	43	0.4361
6	51	0.4322
7	58	0.4316
8	65	0.4294
9	77	0.4294
10	88	0.4294

Optimization started 09-Oct-2012 12:01:35

Optimization converged, 09-Oct-2012 12:03:32

Optimized variable values written to 'DesignVars' in the Design Optimization workspace

[Save Iteration...] [Display Options...] [Optimize]
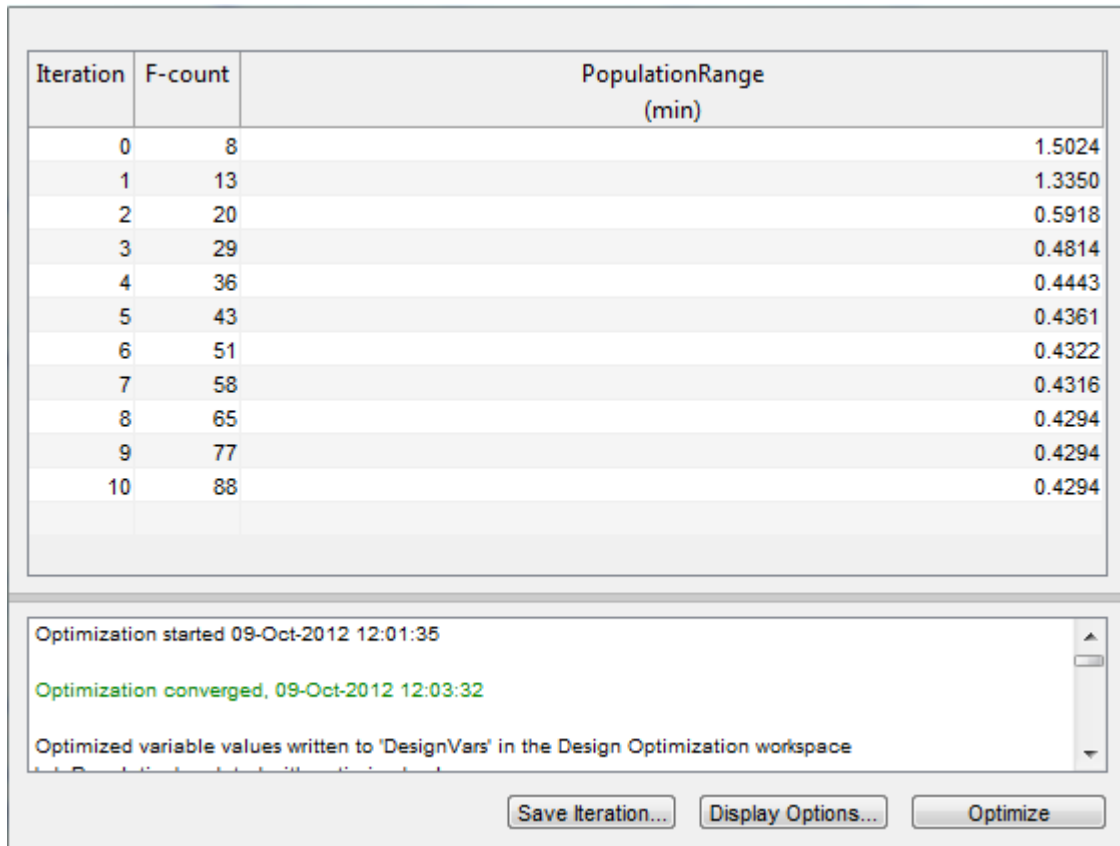
The `P1,P2` plot shows the population dynamics, with the first organism population in blue and the second organism population in red. The dotted lines indicate the population dynamics for different environment capacity values. The `PopulationRange` plot shows that the maximum difference between the two organism populations reduces over time.
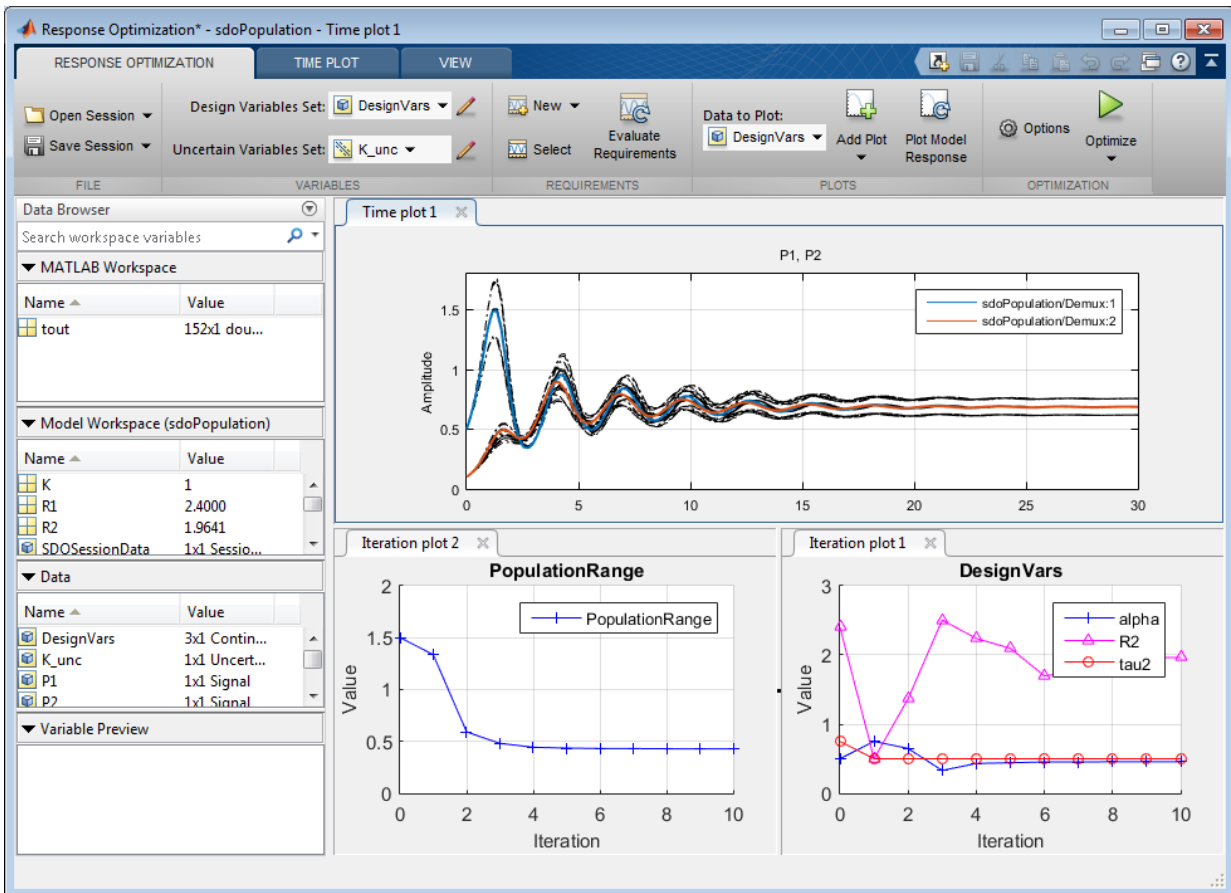
The `Population Phase Portrait` block shows the populations initially varying, but they eventually converge to stable population sizes.
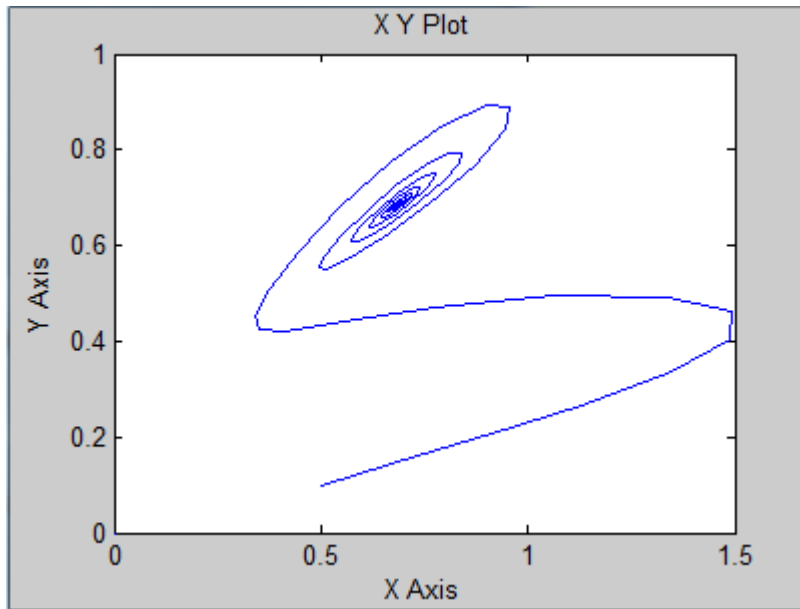
```
% Close the model
bdclose('sdoPopulation')
```

Design Optimization with Uncertain Variables (Code)

This example shows how to optimize a design when there are uncertain variables. You optimize the dimensions of a Continuously Stirred Tank Reactor (CSTR) to minimize product concentration variation and production cost in case of varying, or uncertain, feed stock.

Continuously Stirred Tank Reactor (CSTR) Model

Continuously Stirred Tank Reactors (CSTRs) are common in the process industry. The Simulink model, sdoCSTR, models a jacketed diabatic (i.e., non-adiabatic) tank reactor described in [1]. The CSTR is assumed to be perfectly mixed, with a single first-order exothermic and irreversible reaction, $A \rightarrow B$. A, the reactant, is converted to B, the product.

In this example, you use the following two-state CSTR model, which uses basic accounting and energy conservation principles:

$$\frac{dC_A}{dt} = \frac{F}{A*h}(C_{feed} - C_A) - r*C_A$$

$$\frac{dT}{dt} = \frac{F}{A*h}(T_{feed} - T) - \frac{H}{c_p\rho}r - \frac{U}{c_p*\rho*h}(T - T_{cool})$$

$$r = k_0 * e^{\frac{-E}{R*T}}$$

- C_A, and C_{feed} - Concentrations of A in the CSTR and in the feed [kgmol/m^3]

- T, T_{feed}, and T_{cool} - CSTR, feed, and coolant temperatures [K]

- F and ρ - Volumetric flow rate [m^3/h] and the density of the material in the CSTR [1/m^3]

- h and A - Height [m] and heated cross-sectional area [m^2] of the CSTR.

- k_0 - Pre-exponential non-thermal factor for reaction $A \rightarrow B$ [1/h]

- E and H - Activation energy and heat of reaction for $A \rightarrow B$ [kcal/kgmol]

- R - Boltzmann's gas constant [kcal/(kgmol * K)]

- c_p and U - Heat capacity [kcal/K] and heat transfer coefficients [kcal/(m^2 * K * h)]

Open the Simulink model.

```
open_system('sdoCSTR');
```


Copyright 2012 The MathWorks, Inc.

The model includes a cascaded PID controller in the `Controller` subsystem. The controller regulates the reactor temperature, T, and reactor residual concentration, C_A.

CSTR Design Problem

Assume that the CSTR is cylindrical, with the coolant applied to the base of the cylinder. Tune the CSTR cross-sectional area, A, and CSTR height, h, to meet the following design goals:

- Minimize the variation in residual concentration, C_A. Variations in the residual concentration negatively affect the quality of the CSTR product. Minimizing the variations also improves CSTR profit.

- Minimize the mean coolant temperature T_{cool}. Heating or cooling the jacket coolant temperature is expensive. Minimizing the mean coolant temperature improves CSTR profit.

The design must allow for variations in the quality of supply feed concentration, C_{feed}, and feed temperature, T_{feed}. The CSTR is fed with feed from different suppliers. The quality of the feed differs from supplier to supplier and also varies within each supply batch.

Specify Design Variables

Select the following model parameters as design variables for optimization:

- Cylinder cross-sectional area A

- Cylinder height h

```
p = sdo.getParameterFromModel('sdoCSTR',{'A','h'});
```

Limit the cross-sectional area to a range of [1 2] m^2.

```
p(1).Minimum = 1;
p(1).Maximum = 2;
```

Limit the height to a range of [1 3] m.

```
p(2).Minimum = 1;
p(2).Maximum = 3;
```

Specify Uncertain Variables

Select the feed concentration and feed temperature as uncertain variables. You evaluate the design using different values of feed temperature and concentration.

```
pUnc = sdo.getParameterFromModel('sdoCSTR',{'FeedCon0','FeedTemp0'});
```

Create a parameter space for the uncertain variables. Use normal distributions for both variables. Specify the mean as the current parameter value. Specify a variance of 5% of the mean for the feed concentration and 1% of the mean for the temperature.

```
uSpace = sdo.ParameterSpace(pUnc);
uSpace = setDistribution(uSpace,'FeedCon0',makedist('normal',pUnc(1).Value,0.05*pUnc(1)
uSpace = setDistribution(uSpace,'FeedTemp0',makedist('normal',pUnc(2).Value,0.01*pUnc(2
```

The feed concentration is inversely correlated with the feed temperature. Add this information to the parameter space.

```
%uSpace.RankCorrelation = [1 -0.6; -0.6 1];
```

The rank correlation matrix has a row and column for each parameter with the (i,j) entry specifying the correlation between the i and j parameters.

Sample the parameter space. The scatter plot shows the correlation between concentration and temperature.

```
rng('default'); %For reproducibility
uSmpl = sdo.sample(uSpace,60);
sdo.scatterPlot(uSmpl)
```
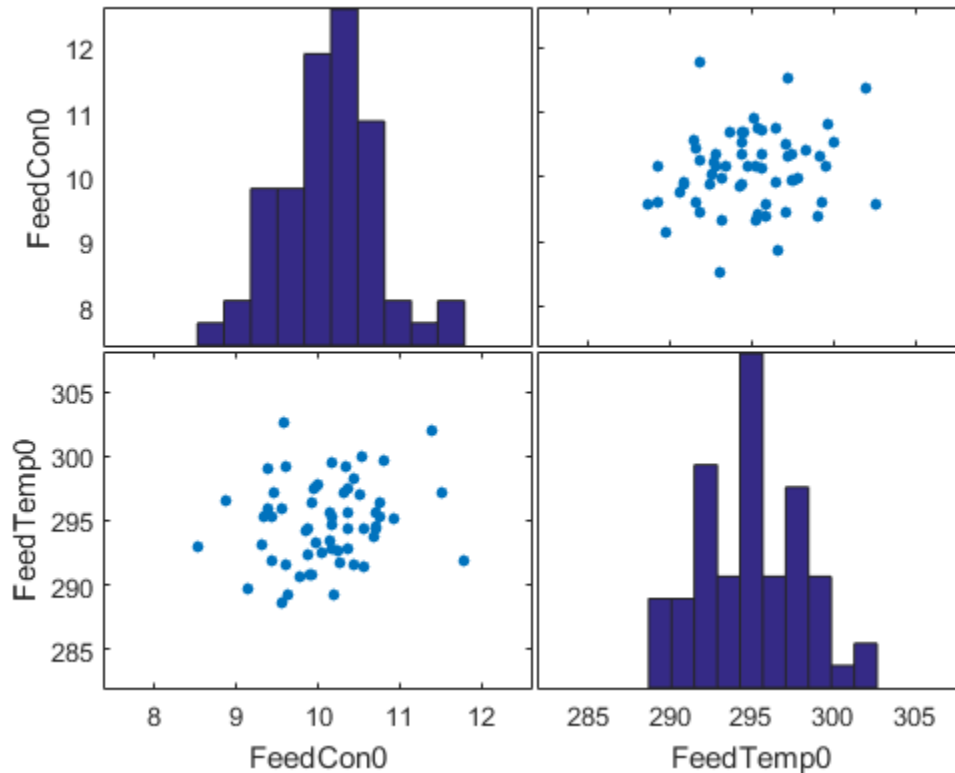
Ideally you want to evaluate the design for every combination of points in the design and uncertain spaces, which implies 30*60 = 1800 simulations. Each simulation takes around 0.5 sec. You can use parallel computing to speed up the evaluation. For this example you instead only use the samples that have maximum & minimum concentration and temperature values, reducing the evaluation time to around 1 min.

```
[~,iminC] = min(uSmpl.FeedCon0);
[~,imaxC] = max(uSmpl.FeedCon0);
[~,iminT] = min(uSmpl.FeedTemp0);
[~,imaxT] = max(uSmpl.FeedTemp0);
uSmpl = uSmpl(unique([iminC,imaxC,iminT,imaxT]) ,:);
```

Specify Design Requirements

The design requirements require logging model signals. During optimization, the model is simulated using the current value of the design variables. Logged signals are used to evaluate the design requirements.

Log the following signals:

- CSTR concentration, available at the second output port of the sdoCSTR/CSTR block

```
Conc = Simulink.SimulationData.SignalLoggingInfo;
Conc.BlockPath             = 'sdoCSTR/CSTR';
Conc.OutputPortIndex       = 2;
Conc.LoggingInfo.NameMode  = 1;
Conc.LoggingInfo.LoggingName = 'Concentration';
```

- Coolant temperature, available at the first output of the sdoCSTR/Controller block

```
Coolant = Simulink.SimulationData.SignalLoggingInfo;
Coolant.BlockPath             = 'sdoCSTR/Controller';
Coolant.OutputPortIndex       = 1;
Coolant.LoggingInfo.NameMode  = 1;
Coolant.LoggingInfo.LoggingName = 'Coolant';
```

Create and configure a simulation test object to log the required signals.

```
simulator = sdo.SimulationTest('sdoCSTR');
simulator.LoggingInfo.Signals = [Conc,Coolant];
```

Create Objective/Constraint Function

Create a function to evaluate the CSTR design. This function is called at each optimization iteration.

Use an anonymous function with one argument that calls the `sdoCSTR_design` function.

```
evalDesign = @(p) sdoCSTR_design(p,simulator,pUnc,uSmpl);
```

The `evalDesign` function:

• Has one input argument that specifies the CSTR dimensions

• Returns the optimization objective value

The `sdoCSTR_design` function uses a `for` loop that iterates through the sample values specified for the feed concentration. Within the loop, the function:

• Simulates the model using the current iterate, feed concentration, and feed temperature values

• Calculates the residual concentration variation and coolant temperature costs

To view the objective function, type `edit sdoCSTR_design`.

Evaluate Initial Design

Call the `evalDesign` function with the initial CSTR dimensions.

```
dInit = evalDesign(p)


dInit =

              F: 11.3358
       costConc: 6.4387
    costCoolant: 4.8971
```

Plot the model response for the initial design. Simulate the model using the sample feed concentration values. The plot shows the variation in the residual concentration and coolant temperature.

```
sdoCSTR_plotModelResponse(p,simulator,pUnc,uSmpl);
```
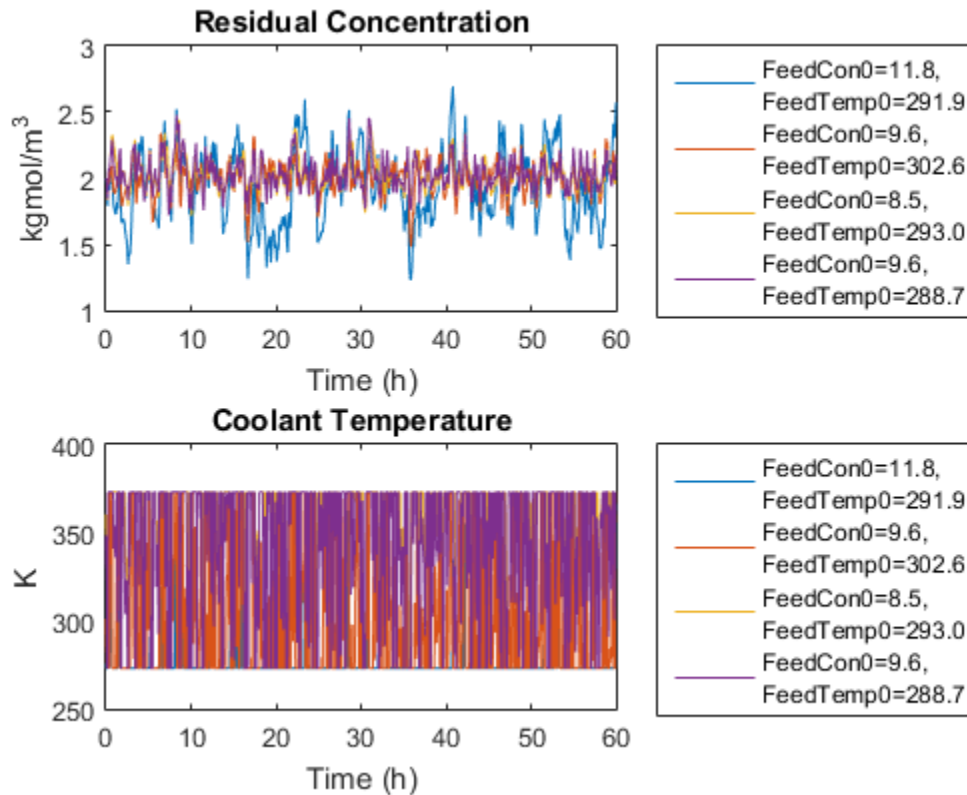
The `sdoCSTR_plotModelResponse` function plots the model response. To view this function, type `edit sdoCSTR_plotModelResponse`.

Optimize Design

Pass the objective function and initial CSTR dimensions to `sdo.optimize`.

```
pOpt = sdo.optimize(evalDesign,p)
```

```
 Optimization started 20-Jan-2015 08:49:38

                                  max        Step-size    First-order
  Iter F-count         f(x)    constraint                  optimality
     0      4       5.17935            0
```

```
 1      8      3.81245          0          2.01         7.81
 2     12      2.65827          0          0.574        3.06
 3     16      2.53697          0          0.162        0.423
 4     20      2.52022          0          0.0154       0.249
 5     24      2.48533          0          0.072        0.163
 6     28      2.47909          0          0.0285       0.107
 7     37      2.47708          0          0.0019       0.143
 8     42      2.46892          0          0.0477       0.452
 9     50      2.46495          0          0.00891      0.397
10     65      2.46444          0          0.00127      0.351
11     72      2.46444          0          0.000801     0.351
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the selected value of the step size tolerance and constraints are
satisfied to within the selected value of the constraint tolerance.

pOpt(1,1) =

       Name: 'A'
      Value: 2
    Minimum: 1
    Maximum: 2
       Free: 1
      Scale: 0.5000
       Info: [1x1 struct]


pOpt(2,1) =

       Name: 'h'
      Value: 2.2093
    Minimum: 1
    Maximum: 3
       Free: 1
      Scale: 2
       Info: [1x1 struct]


2x1 param.Continuous
```

Evaluate Optimized Design

Call the `evalDesign` function with the optimized CSTR dimensions.

```
dFinal = evalDesign(pOpt)


dFinal =

              F: 2.4644
       costConc: 1.4454
    costCoolant: 1.0191
```

Plot the model response for the optimized design. Simulate the model using the sample feed concentration values. The optimized design reduces the residual concentration variation and average coolant temperature for different feed stocks.

```
sdoCSTR_plotModelResponse(pOpt,simulator,pUnc,uSmpl);
```
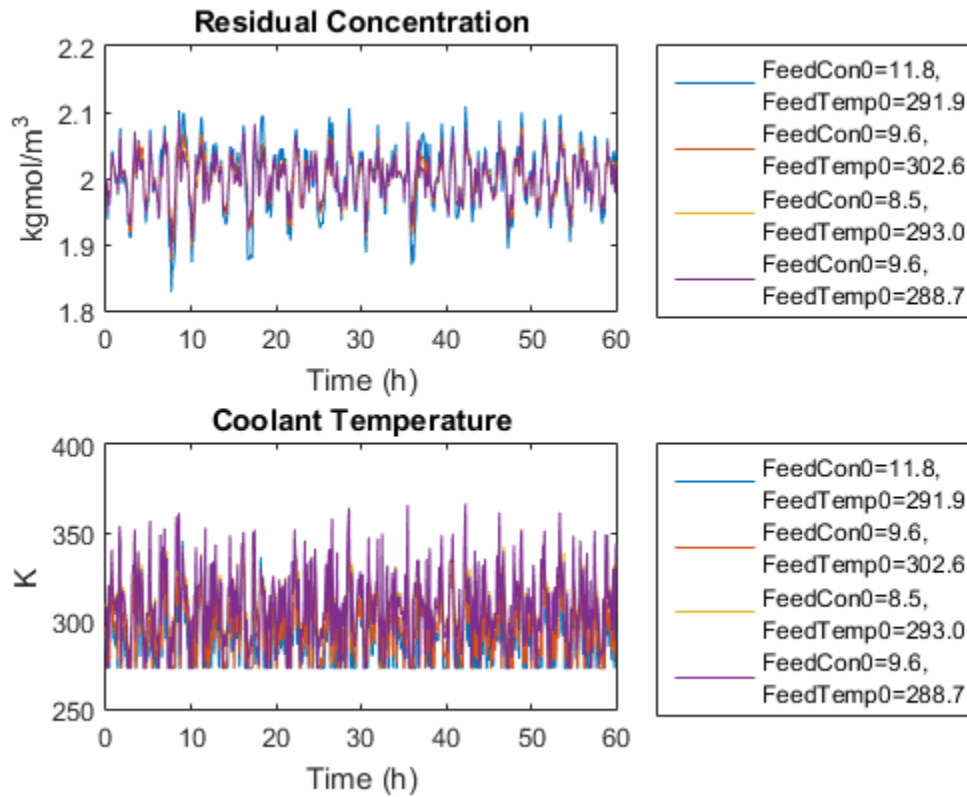
Related Examples

To learn how to use sensitivity analysis to explore the CSTR design space and select an initial design for optimization, see "Design Optimization with Uncertain Variables (Code)".

References

[1] Bequette, B.W. *Process Dynamics: Modeling, Analysis and Simulation.* 1st ed. Upper Saddle River, NJ: Prentice Hall, 1998.

```
% Close the model
bdclose('sdoCSTR')
```

Generate MATLAB Code for Design Optimization Problems (GUI)

This example shows how to automatically generate a MATLAB function to solve a Design Optimization problem. You use the Response Optimization tool to define an optimization problem for a hydraulic cylinder design and generate MATLAB code to solve this optimization problem.

Hydraulic Cylinder Design Problem

The "Design Optimization to Meet a Custom Objective Using the Response Optimization Tool" example shows how to use the Response Optimization tool to optimize a cylinder design. In this example we load a pre-configured Response Optimization tool session based on that example.

```
load sdoHydraulicCylinder_sdosession
sdotool(SDOSessionData)
```


Single Hydraulic Cylinder Simulation
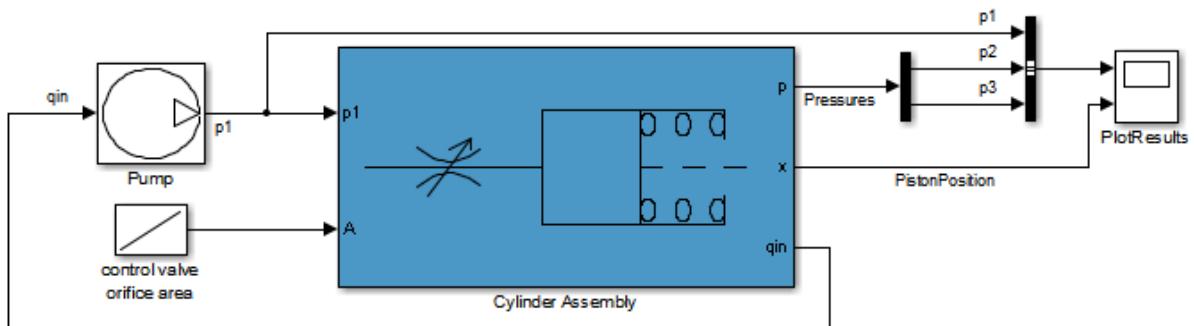
Copyright 1990-2011 The MathWorks, Inc.
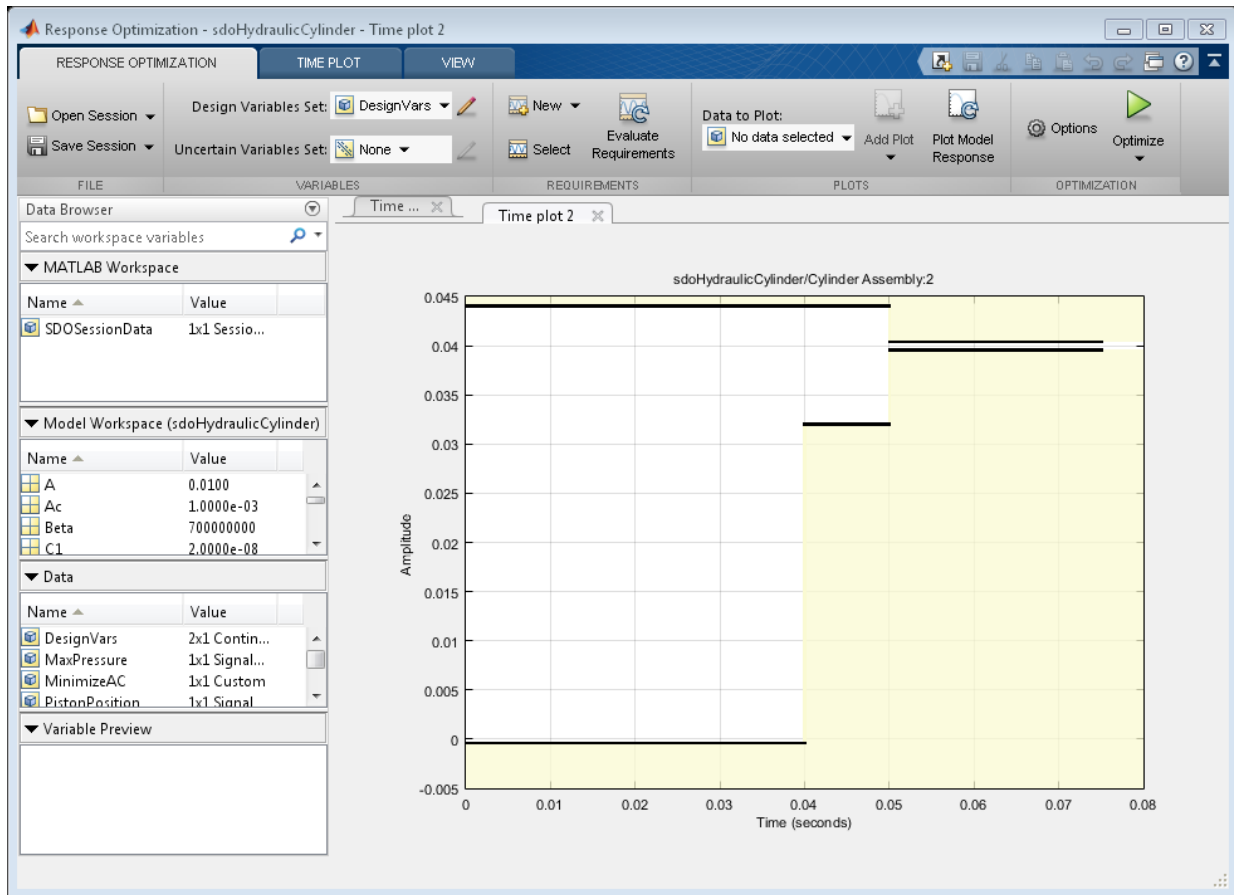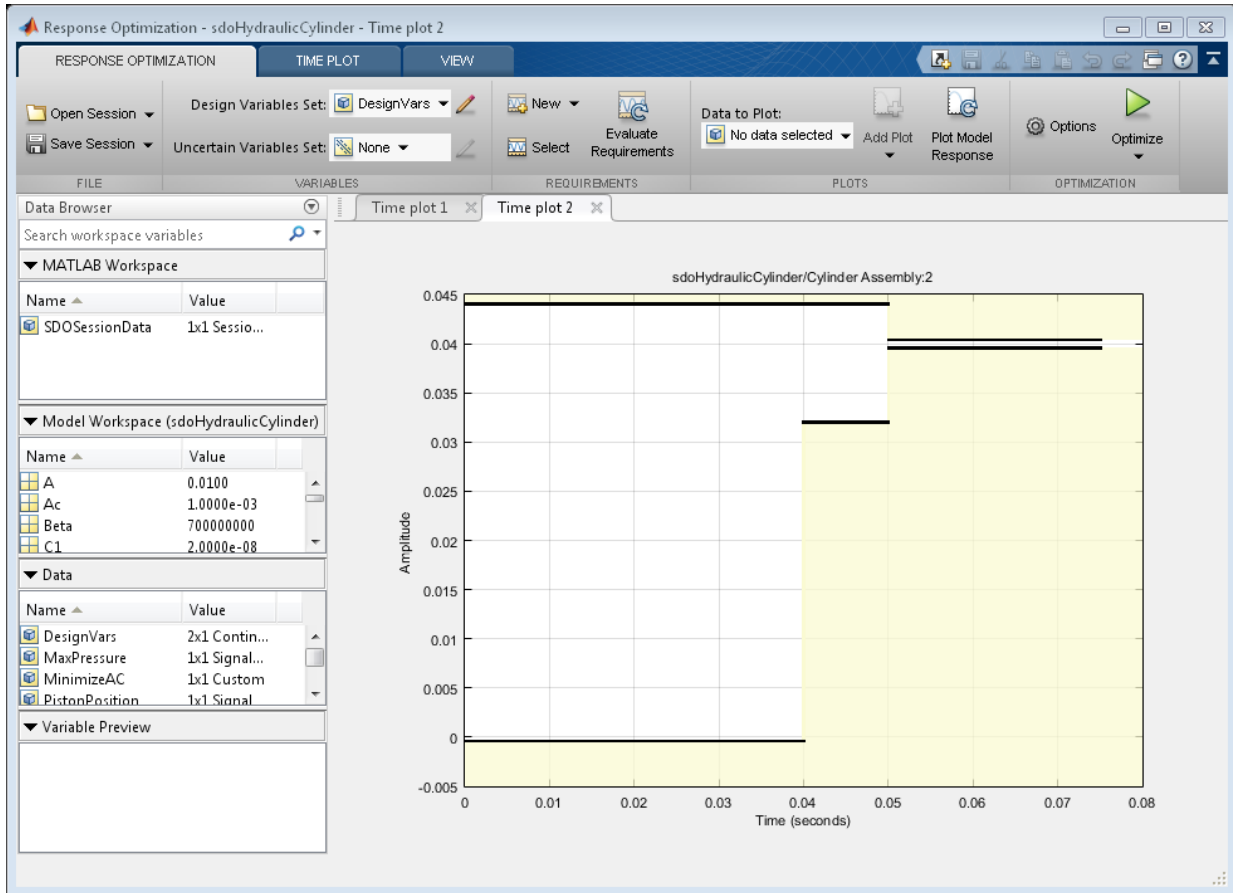
Generate MATLAB Code

From the **Optimize** list, select **Generate MATLAB Code**.
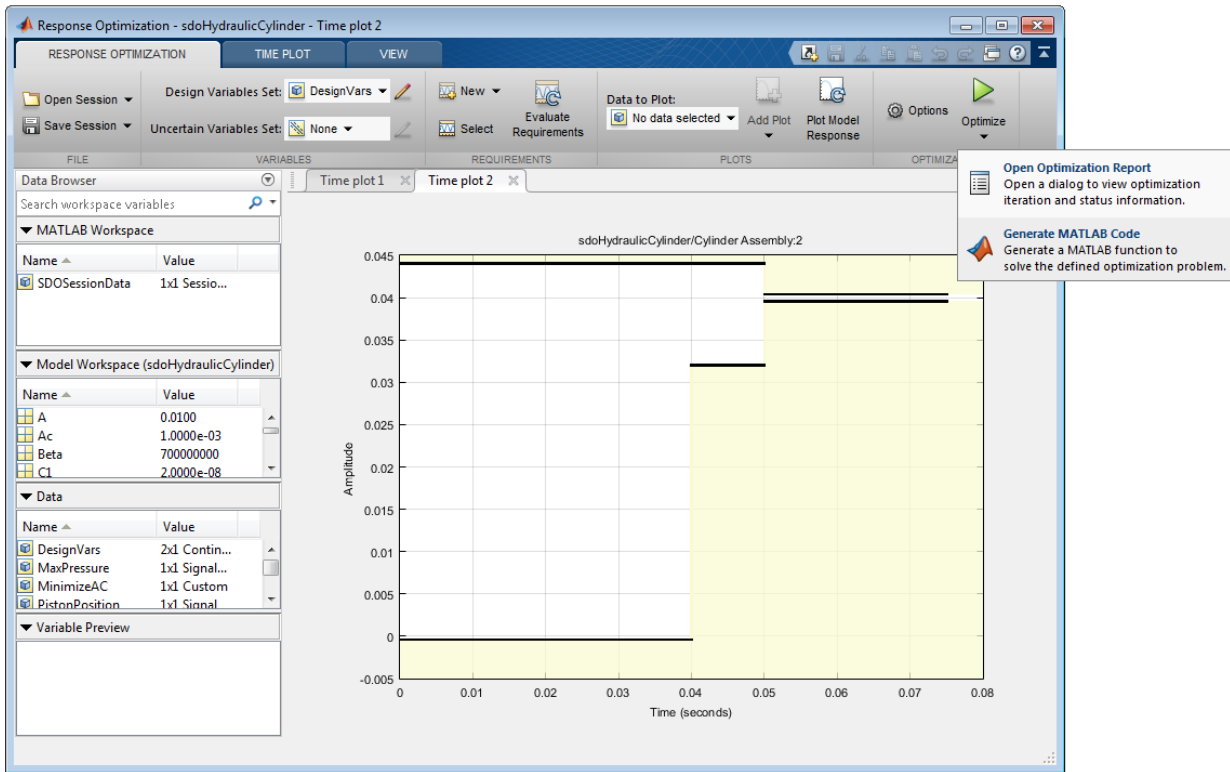
The generated code is added to the MATLAB editor as an unsaved MATLAB function.

```
Editor - Untitled29*                                                        ⊙ ×
 1   ☐ function [Optimized_DesignVars, Info] = sdo_sdoHydraulicCylinder(DesignVars)
 2   ☐ %SDO_SDOHYDRAULICCYLINDER
 3     %
 4     % Solve a design optimization problem for the sdoHydraulicCylinder model.
 5     %
 6     % The function returns optimized parameter values, Optimized_DesignVars,
 7     % and optimization termination information, Info.
 8     %
 9     % The, DesignVars, input argument defines the model parameters to optimize,
10     % if omitted the parameters specified in the function body are optimized.
11     %
12     % Modify the function to include or exclude new design requirements or
13     % change the optimization options.
14     %
15     % Auto-generated by SDOTOOL on 15-Mar-2012 13:14:45.
16   └ %
17
18     %% Open the model.
19     open_system('sdoHydraulicCylinder')
20
21     %% Specify Design Variables
22     %
23     % Specify model parameters as design variables to optimize.
24     if nargin < 1 || isempty(DesignVars)
25         DesignVars = sdo.getParameterFromModel('sdoHydraulicCylinder',{'Ac','K'});
26         DesignVars(1).Minimum = 0.0003;
27         DesignVars(1).Maximum = 0.0013;
28         DesignVars(1).Scale = 0.001;
```
```
Untitled29*  ×
```

Examine the generated code. Significant code portions are:
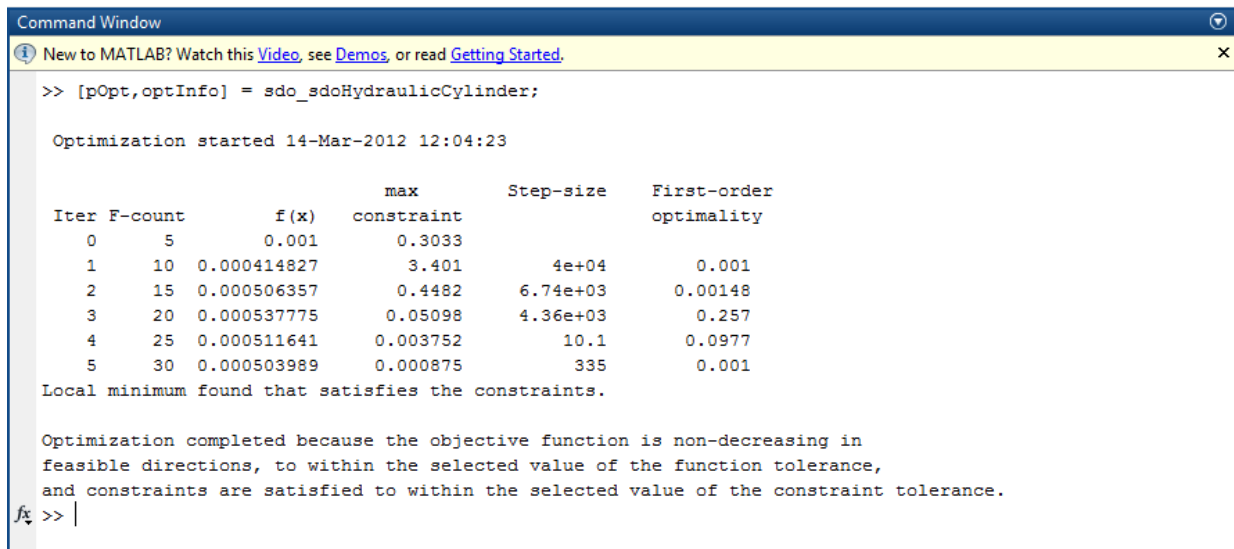
- **Specify Design Variables** - Definition of the model parameters being optimized.

- **Specify Design Requirements** - Definition of the design requirements.

- **Create Optimization Objective Function** - Creation of an anonymous function that calls the subfunction sdoHydraulicCylinder_optFcn, which evaluates the cylinder design. sdo.optimize calls the anonymous function at each iteration.

- **Evaluate custom parameter requirement functions** - Evaluates the custom requirement, MinimizeAC, that uses the sdoHydraulicCylinder_customObjective function.

- **Optimize the Design** - Optimization using the `sdo.optimize` command.

Select **Save** from the MATLAB editor to save the generated function.

Run Generated Code

Run the generated function.

```
Command Window

New to MATLAB? Watch this Video, see Demos, or read Getting Started.

>> [pOpt,optInfo] = sdo_sdoHydraulicCylinder;

 Optimization started 14-Mar-2012 12:04:23

                              max       Step-size    First-order
 Iter F-count         f(x)    constraint              optimality
    0      5        0.001       0.3033
    1     10  0.000414827        3.401        4e+04        0.001
    2     15  0.000506357       0.4482     6.74e+03      0.00148
    3     20  0.000537775      0.05098     4.36e+03        0.257
    4     25  0.000511641     0.003752         10.1       0.0977
    5     30  0.000503989     0.000875          335        0.001
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the selected value of the function tolerance,
and constraints are satisfied to within the selected value of the constraint tolerance.

fx >> |
```

The first output argument, `pOpt`, contains the optimized parameter values and the second output argument, `optInfo`, contains optimization information.

Modify the Generated Code

You can:

- Modify the generated `sdo_sdoHydraulicCylinder` function to include or exclude new design requirements or change the optimization options.

- Call the generated `sdo_sdoHydraulicCylinder` function with a different set of parameters to optimize.

For details on how to write an objective/constraint function to use with the `sdo.optimize` command, type `help sdoExampleCostFunction` at the MATLAB command prompt.

Close the model

```
delete(sdotool('sdoHydraulicCylinder'))
bdclose('sdoHydraulicCylinder')
```

Skip Model Simulation Based on Parameter Constraint Violation (GUI)

This example shows how to optimize a design and specify parameter-only constraints that prevent the model from being evaluated in an invalid solution space.

During optimization, the solver may try a design variable set that results in a model simulation error, which can be computationally expensive. If you can define a parameter-only constraint that identifies such a design variable set, then the solver can use the constraint to skip such sets. In other words, you can configure the optimization to be more efficient by disallowing design variable sets that lead to simulation errors.

In this example, you optimize thermostat settings to minimize temperature set-point deviations while satisfying some constraints. One of the constraints applies to the model parameters that define the thermostat switch on/switch off points. If the switch-off point is greater than the switch-on point, evaluating the model leads to a simulation error.

Thermostat Model

Open the model.

```
open_system('sdoThermostat');
```

Thermostat Control

Copyright 2012 The MathWorks, Inc.
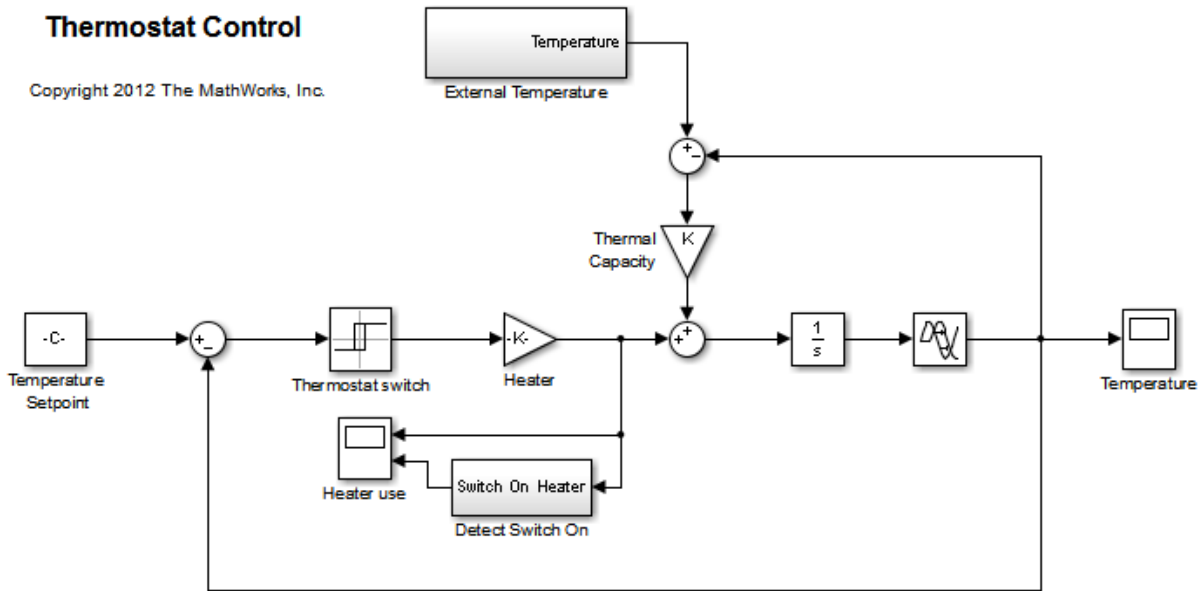
The model describes a simple heater & thermostat that regulate the temperature of a room. The room is subject to external temperature fluctuations. The room temperature is computed using a first-order heat-flow equation:

$$\frac{dT}{dt} = K(T_e - T) + Q$$

Where:

- T is the room temperature (C).

- T_e the external temperature (C).

- Q the heat supplied by the heater (W).

- K the room thermal capacity (J/C).

The heater is controlled by a thermostat that turns on when the difference between the room temperature and temperature set-point exceeds a threshold. The heater turns off when the error drops below a threshold.

The heater operation is displayed in the `Heater use` scope. The upper axis is the delivered heat and the lower axis shows the times when the heater is switched on.
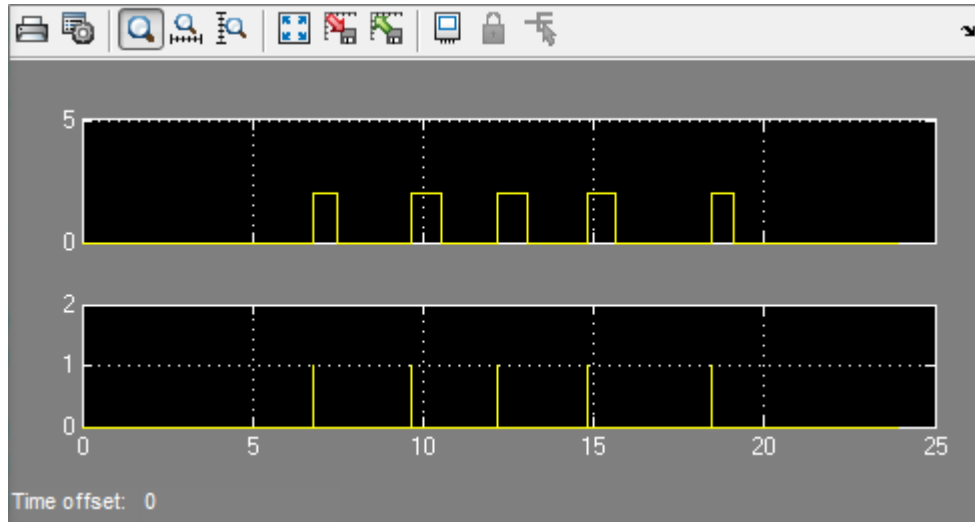
The room temperature is displayed in the `Temperature` scope.
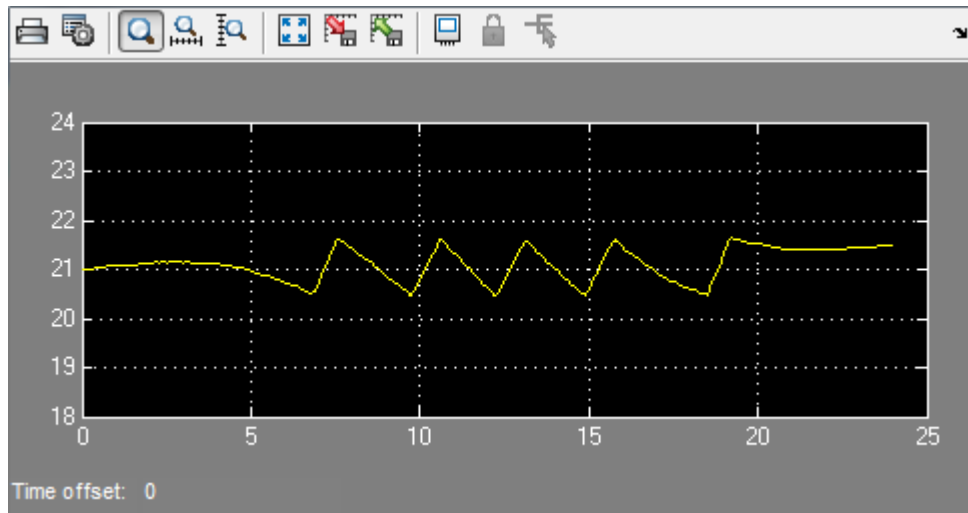
Thermostat Design Problem

You tune the thermostat turn-on and turn-off temperature thresholds, and also the heater power. The Thermostat switch block specifies the turn-on and turn-off thresholds using the variables H_on and H_off. The Heater block specifies the heater power using the variable Hgain.

The design requirements are:

- Minimize the difference between the room temperature and temperature set-point over a 24 hour period.

- The heater must not turn on more than 12 times during the 24 hour period.

- The thermostat turn-on temperature must be greater than the thermostat turn-off temperature. If this constraint is violated, the model is invalid and cannot be simulated or evaluated.

Open the Response Optimization Tool

Open a pre-configured Response Optimization tool session.

```
load sdoThermostat_sdosession
sdotool(SDOSessionData)
```
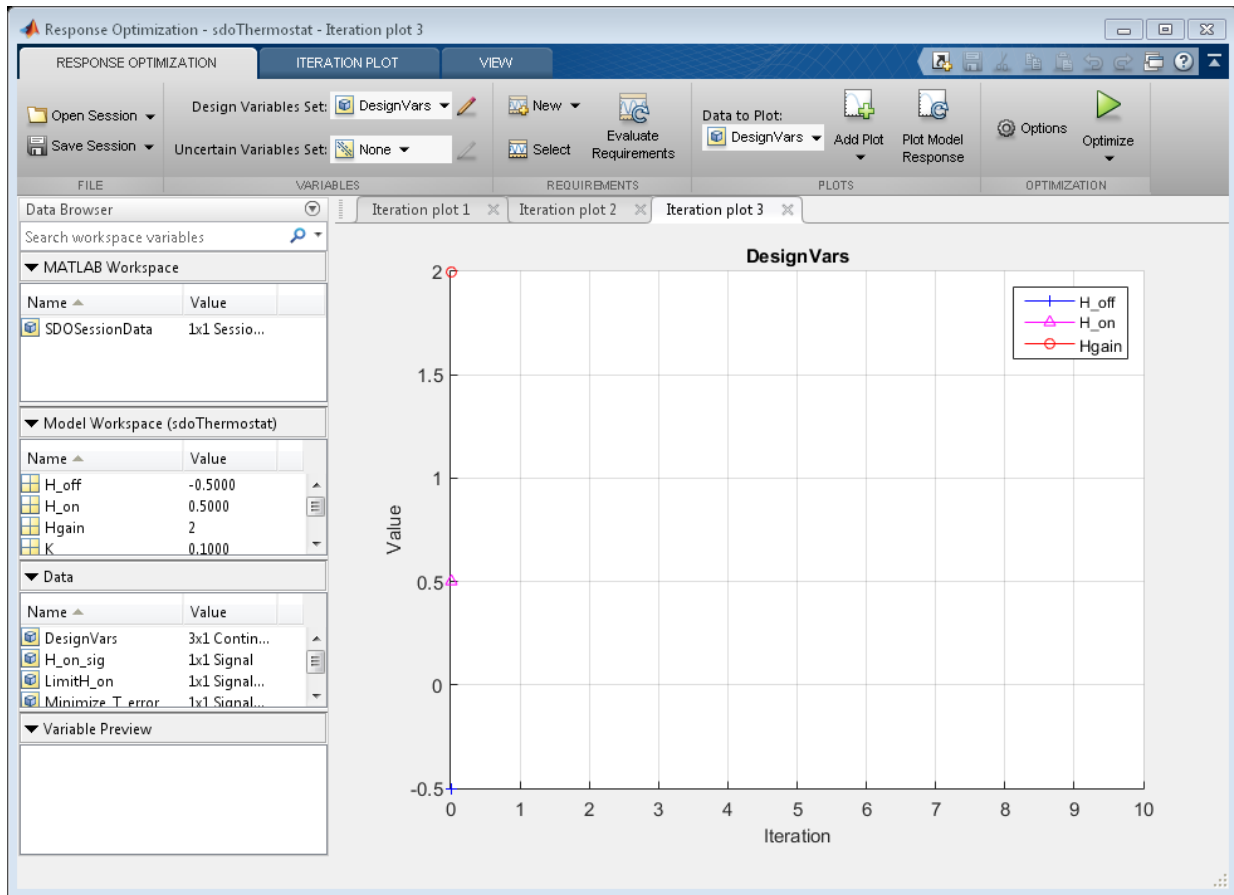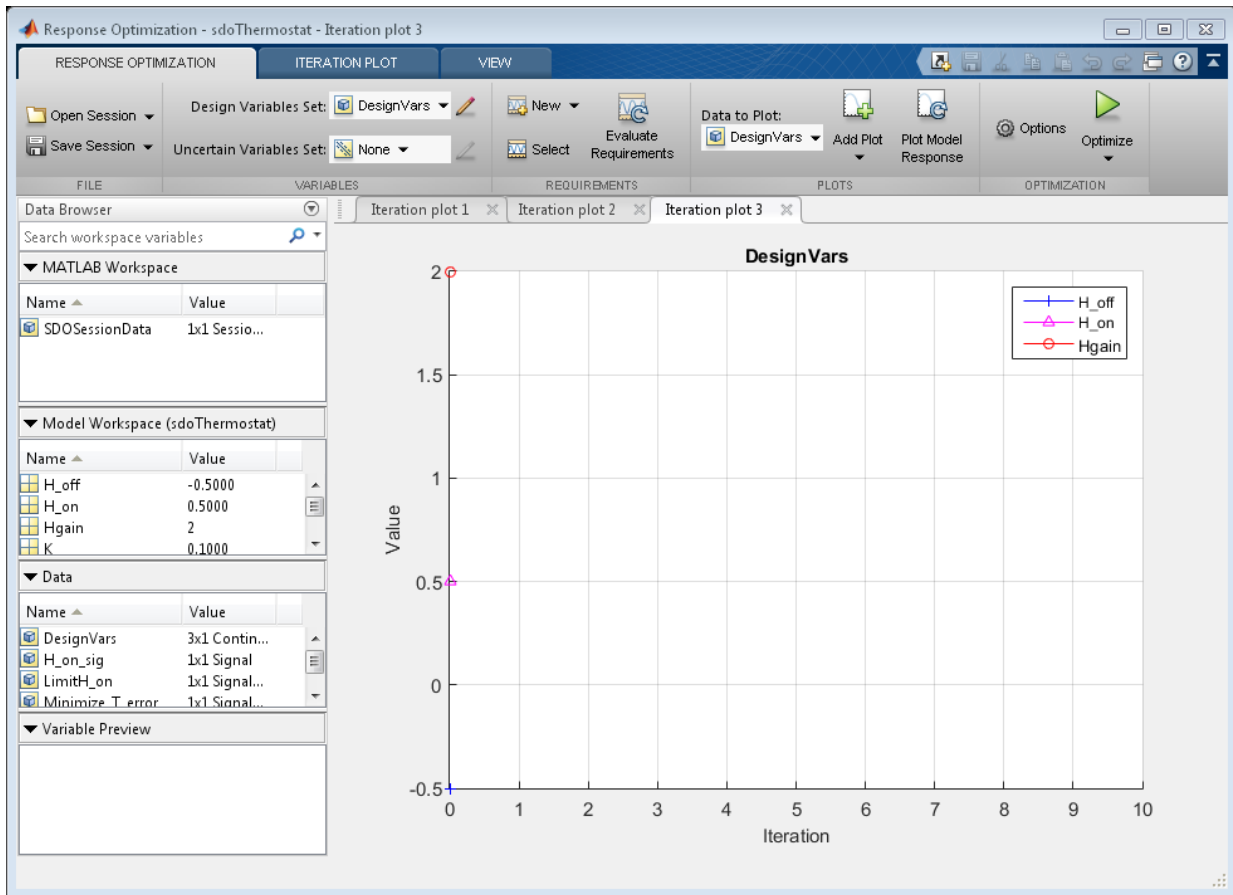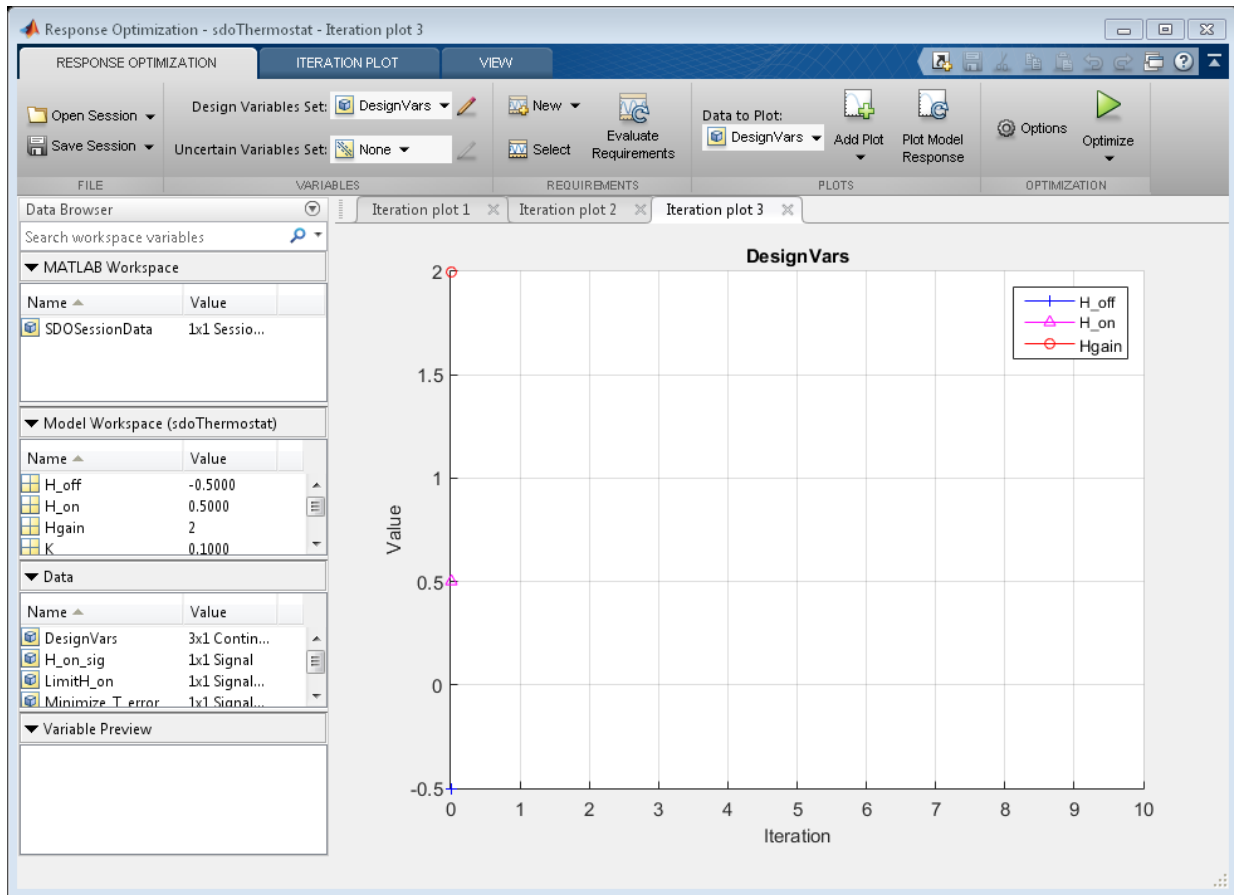
The pre-configured session specifies the following variables:

- DesignVars - Design variables set for the H_on, H_off, and Hgain model parameters.

- Minimize_T_error - Requirement to minimize the temperature deviation from the set-point.

- LimitH_on - Requirement to limit the number of times the thermostat is turned on.

- H_on_sig and T_error - Logged signals. H_on_sig represents when the heater is on. T_error is the difference between the room temperature and the set-point.

Specify Parameter Constraint

The H_on > H_off requirement is not yet defined. Use a custom requirement to specify this constraint and configure the requirement to error if it is not satisfied.

In the **New** drop-down list, select **Custom Requirement**. The Create Requirement dialog opens.

In this dialog, specify the following:

- **Name** - SwitchConstraint.

- **Type** - Select Constrain the function output to be >= 0 from the **Type** list.

- **Function** - @sdoThermostat_SwitchingConstraint.

- **Error if constraint is violated** - Select this check box.
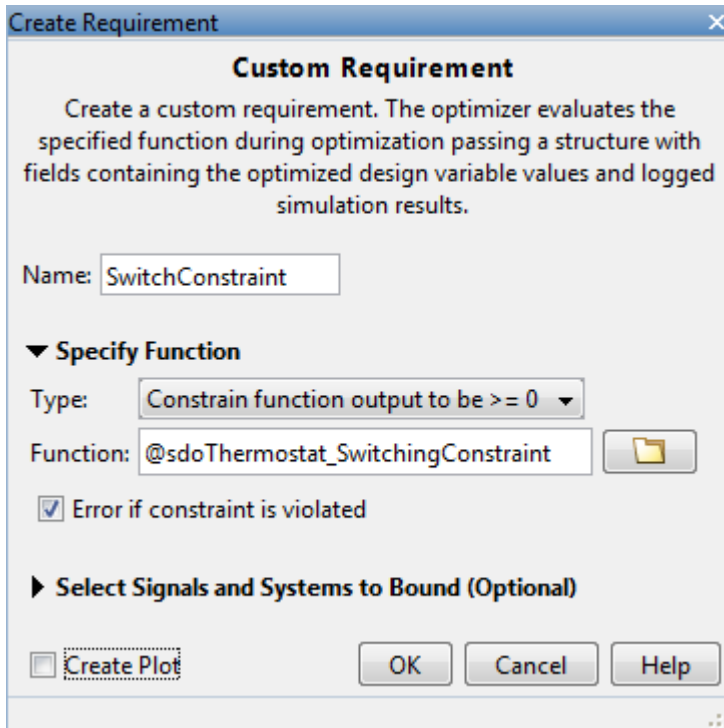
The software calls the `sdoThermostat_SwitchingConstraint` function at each optimization iteration with a structure containing all the design variables. The output of the `sdoThermostat_SwitchingConstraint` function is the difference between the `H_on` and `H_off` values. This difference must be positive for the requirement to be satisfied.

The software evaluates custom requirements that test parameter-only constraints, such as `SwitchConstraint`, before simulating the model and evaluating the remaining requirements.

- If the constraint is violated while the **Error if constraint is violated** check box is *selected*, the software does not simulate the model to evaluate the remaining requirements. Instead, the solver assigns the cost function a `NaN` value for this iteration, evaluates the terminating conditions, and continues.

- If the constraint is violated while the **Error if constraint is violated** check box is *cleared*, the solver will attempt to simulate the model to evaluate the remaining requirements. Simulating the model may lead to a hard error; for example, simulating the thermostat model when `SwitchConstraint` is violated will lead to an error. In this case, the solver assigns the cost function a `NaN` value for this iteration, evaluates the terminating conditions, and continues.

To examine the constraint function, type `edit sdoThermostat_SwitchingConstraint`. The requirement that `H_on` > `H_off` is implemented as `H_on` - `H_off` > 0

Optimize the Design
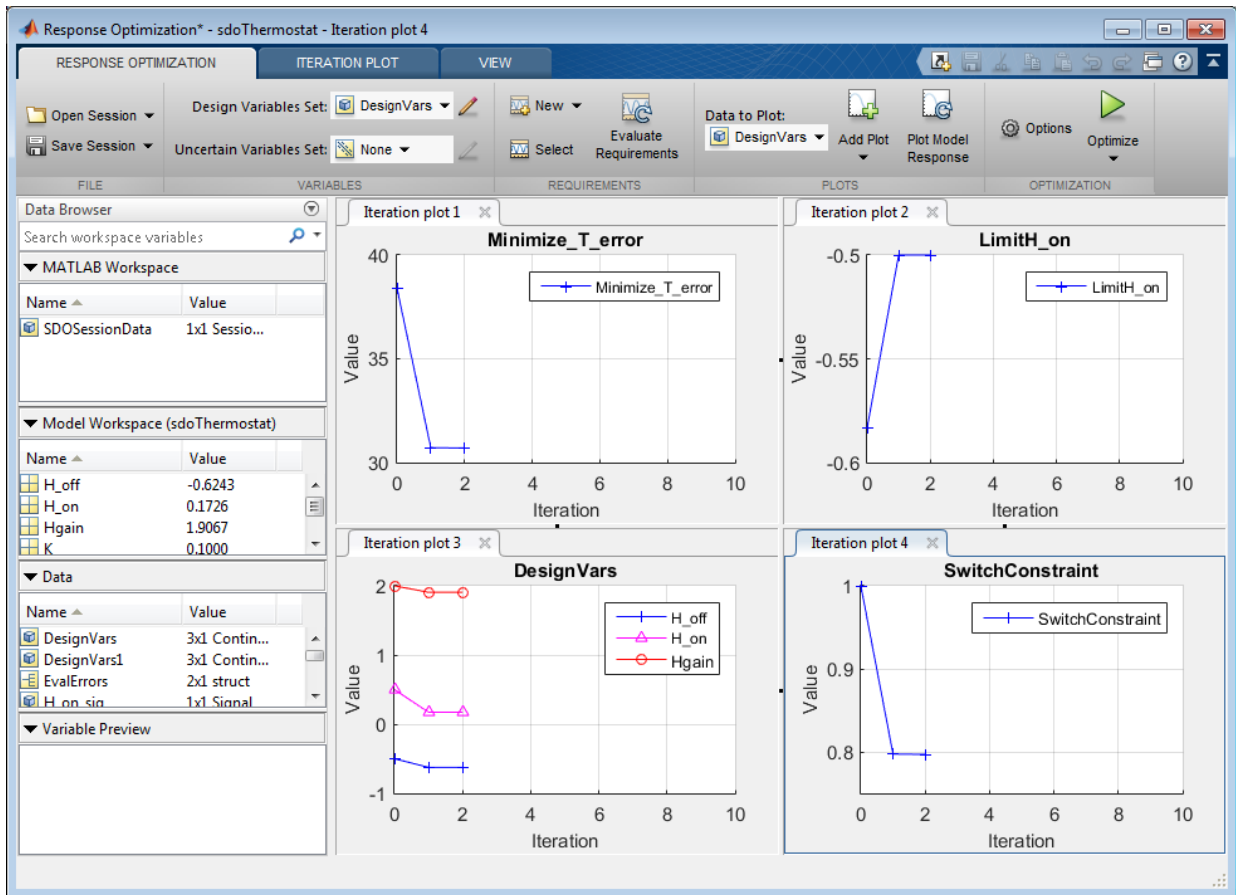
Click **Optimize**.

Optimization Progress Report

Iteration	F-count	SwitchConstraint (>=0)	Minimize_T_error (Minimize)	LimitH_on (<=0)
0	7	1	38.4084	-0.5833
1	28	0.7973	30.7171	-0.5000
2	133	0.7969	30.7003	-0.5000

Optimization started 24-Oct-2014 15:57:03

Optimization converged, 24-Oct-2014 15:57:27
The optimizer encountered 2 errors during the optimization. Details of the errors have been written to 'EvalErrors' in the Design Optimization workspace.

Save Iteration...   Display Options...   Optimize

The Optimization Progress window appears and updates at each iteration. The optimization successfully minimizes the temperature error while satisfying the switching constraints.

During this optimization, the H_on and H_off values never approach the H_on > H_off constraint boundary. So, there is never a danger of violating the constraint. However, changing the optimization algorithm may produce different behavior. For example, changing the optimization algorithm from the one used here, 'Interior-Point', to 'Active-Set' results in H_on and H_off values that are at the constraint boundary. This violation

triggers the `SwitchConstraint` requirement and prevents model simulation for the relevant iterations.

View Optimized Model Response

Simulate the model with the optimized thermostat settings. The optimized heater operation is displayed in the `Heater use` scope where the upper axis is the delivered heat and the lower axis the heater switch on times.
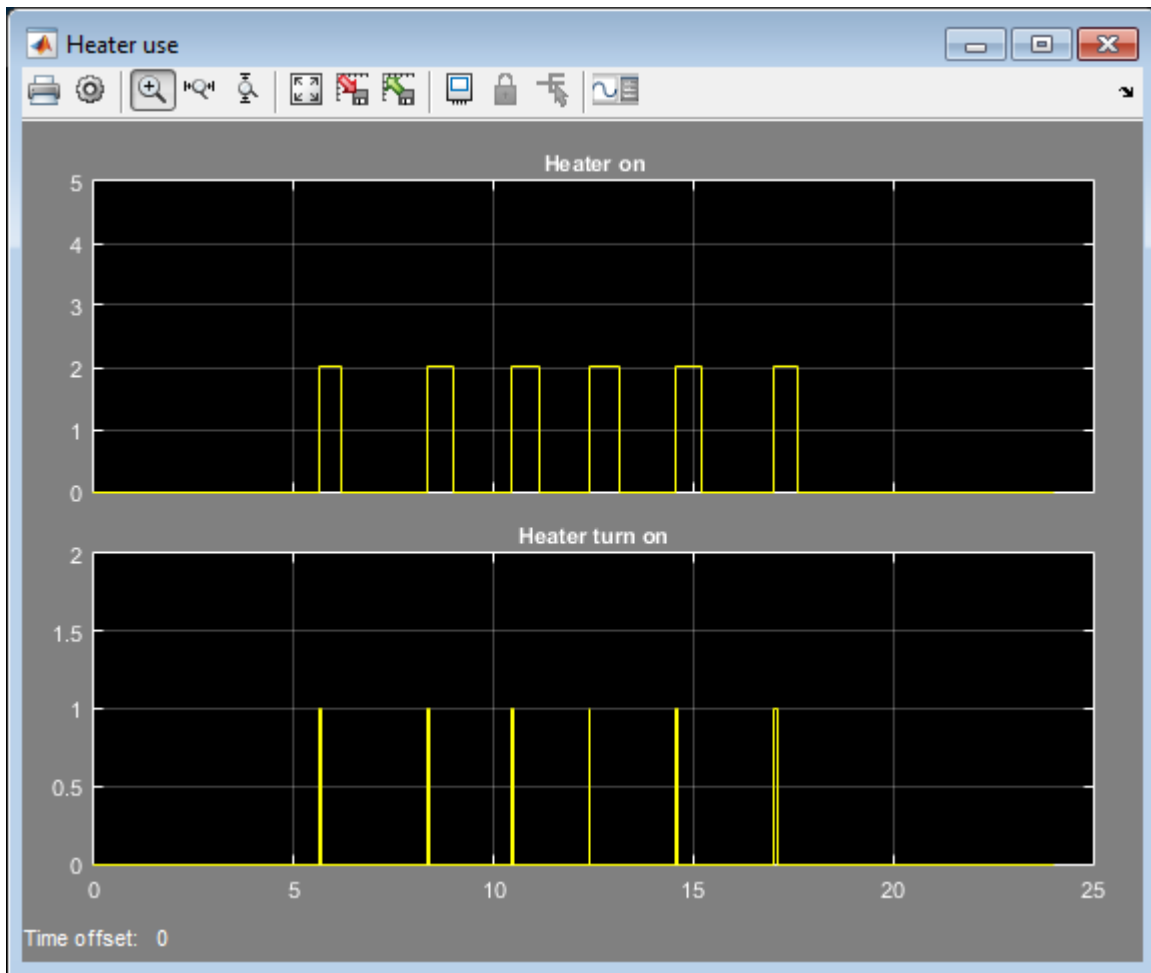
The optimized room temperature is displayed in the `Temperature` scope.
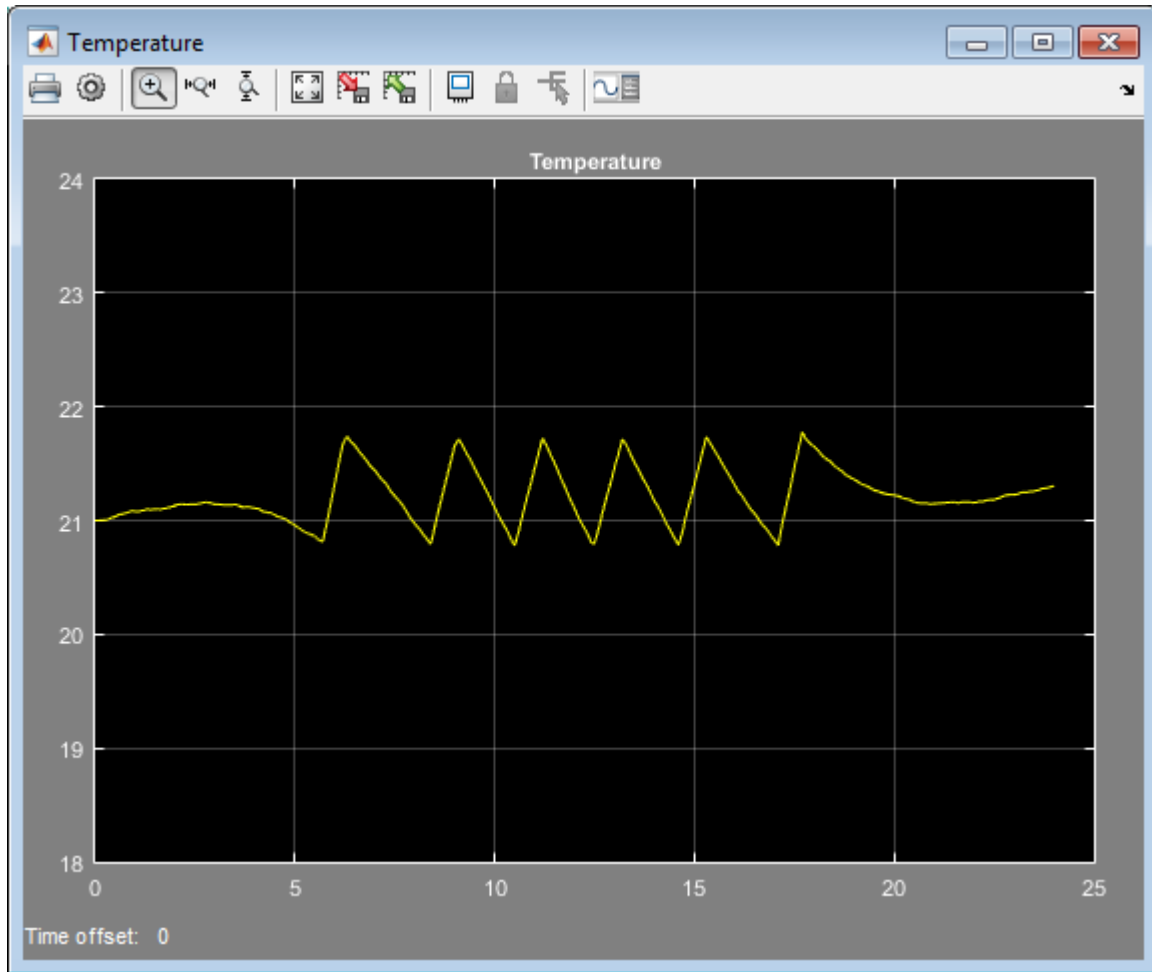
Close the model

```
delete(sdotool('sdoThermostat'))
bdclose('sdoThermostat')
```

Optimizing Parameters for Robustness

In this section...
"What Is Robustness?" on page 3-201
"Sampling Methods for Uncertain Parameters" on page 3-202
"Optimize Parameters for Robustness (GUI)" on page 3-206

What Is Robustness?

A design is *robust* when it's response does not violate design requirements under model parameter variations. Your model may contain parameters whose values are not precisely known. Such parameters vary over a given range of values and are defined as *uncertain parameters*. You may know the nominal value and the range of values in which these uncertain parameters vary.

You can use Simulink Design Optimization software to incorporate the parameter uncertainty to test the robustness of your design. When you optimize parameters for robustness, the optimization solver uses the responses computed using all the uncertain parameter values to adjust the design variable values.

You can specify the same parameter both as a design *and* uncertain variable. However, you cannot use a parameter both as a design and uncertain variable in the same optimization run. Also, you cannot add uncertainty to controller or plant parameters during optimization-based control design in the SISO Design Tool.

The uncertain variables can be scalar, vector, matrix or an expression.

You can test and optimize parameters for model robustness in the following ways:

- **Before Optimization**. Specify the parameter uncertainty *before* you optimize the parameters to meet the design requirements. In this case, the optimization method optimizes the signals based on both nominal parameter values as well as the uncertain values. This mode requires more computational time.

- **After Optimization**. Specify the parameter uncertainty *after* you have optimized the model parameters to meet design requirements. You can then test the effect of the uncertain parameters by plotting the model's response. If the response violates the design requirements, you can optimize the parameters again by including the parameter uncertainty during the optimization.

Related Examples

"Optimize Parameters for Robustness (GUI)" on page 3-206

More About

"Sampling Methods for Uncertain Parameters" on page 3-202

Sampling Methods for Uncertain Parameters

Sample values for uncertain parameters are a vector of numerical values. You can specify the vector yourself or generate a vector of random numbers using the software. The sample values you specify can be uniformly distributed or random. For example, four sample values for two uncertain parameters a and b in the range [0 3] and [1 2.5] may look like the following figure.
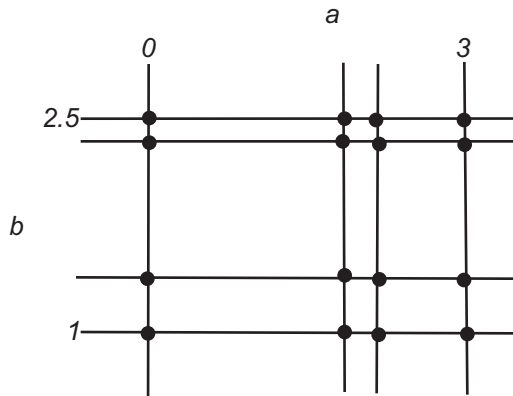
There are two methods to determine the number of sample values to use during optimization:

- Only the combination of minimum and maximum values (circled)
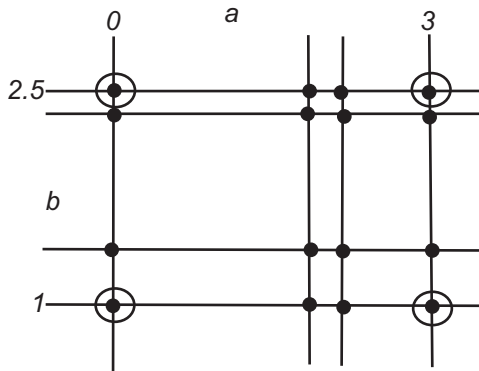
- Combination of the entire set of values (all solid dots in the previous figure)

Tip Using only the minimum and maximum values during optimization increases the computation speed when compared to using the entire set of values.

For the previous example, there are 4 combinations using the minimum and maximum values and 16 combinations if you use all sample values.

In the Response Optimization tool, you specify the sampling method using the options as shown in the following figure.
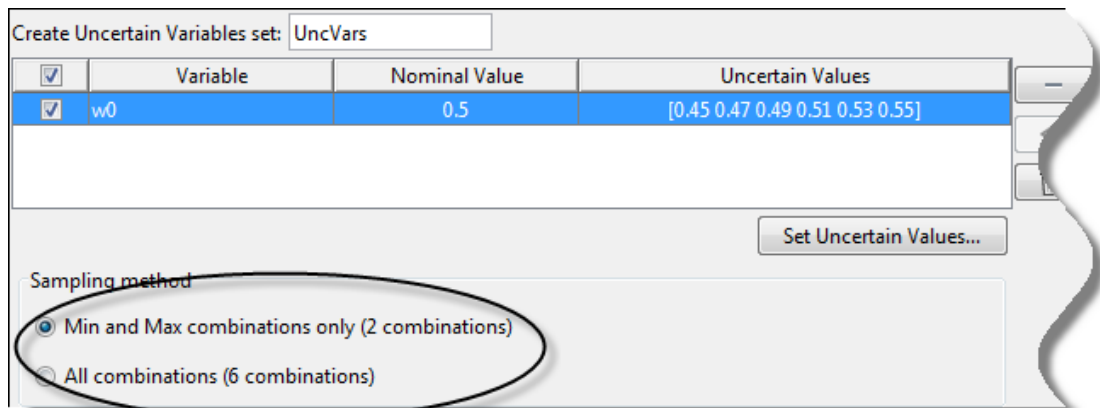
Related Examples

"Optimize Parameters for Robustness (GUI)" on page 3-206

More About

• "What Is Robustness?" on page 3-201

Optimize Parameters for Robustness (GUI)

This example shows how to optimize parameters for model robustness.

1   Load a saved Response Optimization tool session.

```
load sldo_model1_desreq_optim_sdosession;
sdotool(SDOSessionData);
```

The sdotool command opens the following Simulink model and a saved Response Optimization tool session.
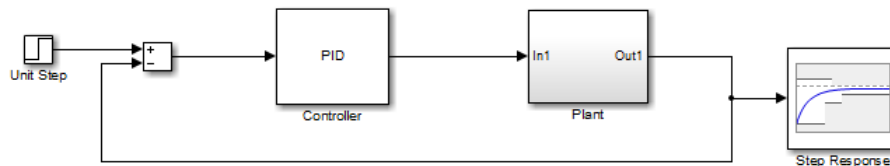
The parameters of this model, Kp, Ki and Kd, have already been optimized to meet the following step response requirements:

- Maximum overshoot of 5%
- Maximum rise time of 10 seconds
- Maximum settling time of 30 seconds

2   Specify parameter uncertainty.

a   In the **Uncertain Variables Set** drop-down list, select **New**.
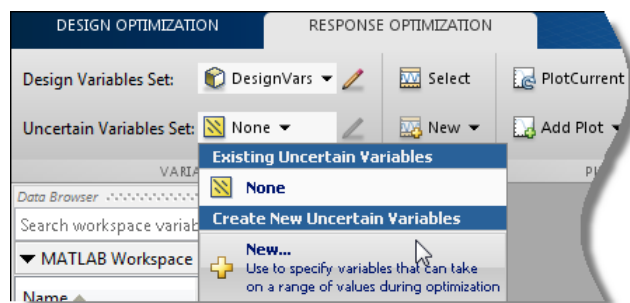
A window opens where you specify uncertain variables.
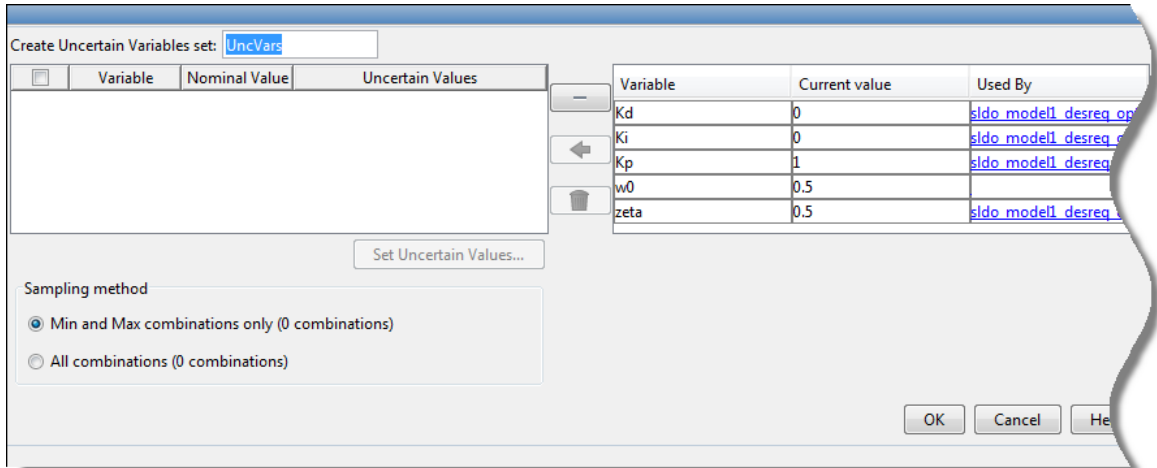
b   Click w0 and zeta to select them.

c
Click to add the selected parameters to an uncertain variables set.
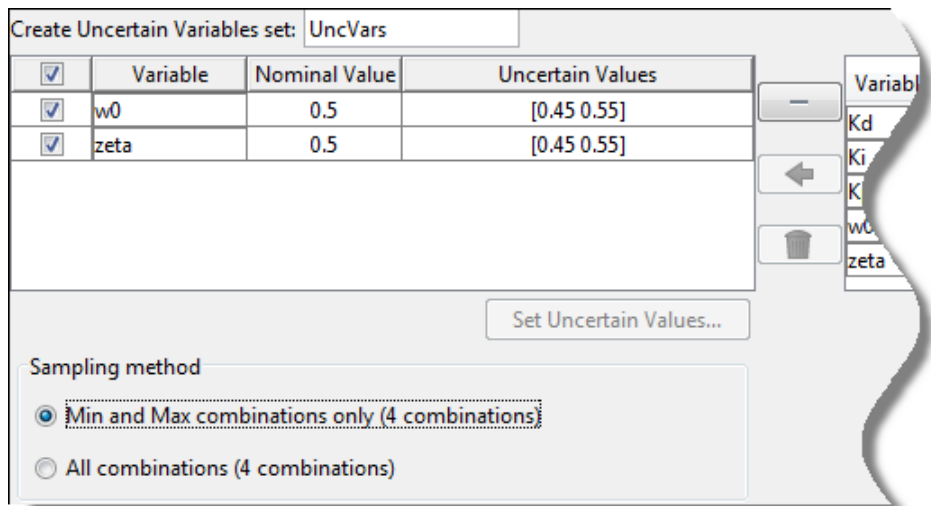
The software displays the following parameter settings:

- **Variable** — Parameter name
- **Nominal Value** — Nominal value of the parameters as specified in the Simulink model
- **Uncertain Values** — Values that the uncertain parameter can take. By default, the maximum and minimum values vary by 10% of the nominal value.

  The total number of sample values to use during optimization is a combination of the maximum and minimum values of the uncertain parameters.

The check-box indicates that the parameter is included in the uncertain variable set. The default uncertain variable set name is `UncVars`.

Click **OK**. A new variable `UncVars` appears in **Data** area of the Response Optimization tool.

Specify Random Values

Instead of specifying sample values, you can auto-generate random values in a specific range. Select a parameter and click **Set Uncertain Values**.
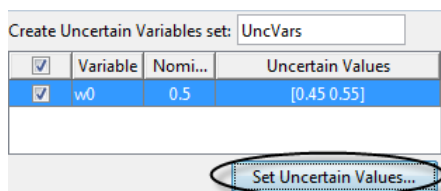
A window opens where you specify the range and the number of samples.
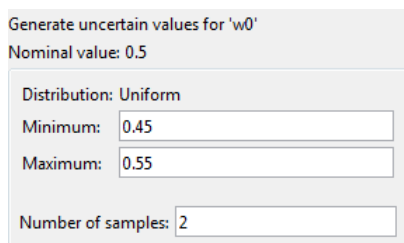
3 Test the model robustness to the uncertain parameters.

    a Click **Plot Current Response**.
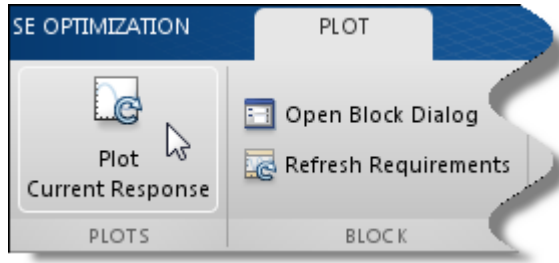
    The step response plot, displaying the requirements, updates.
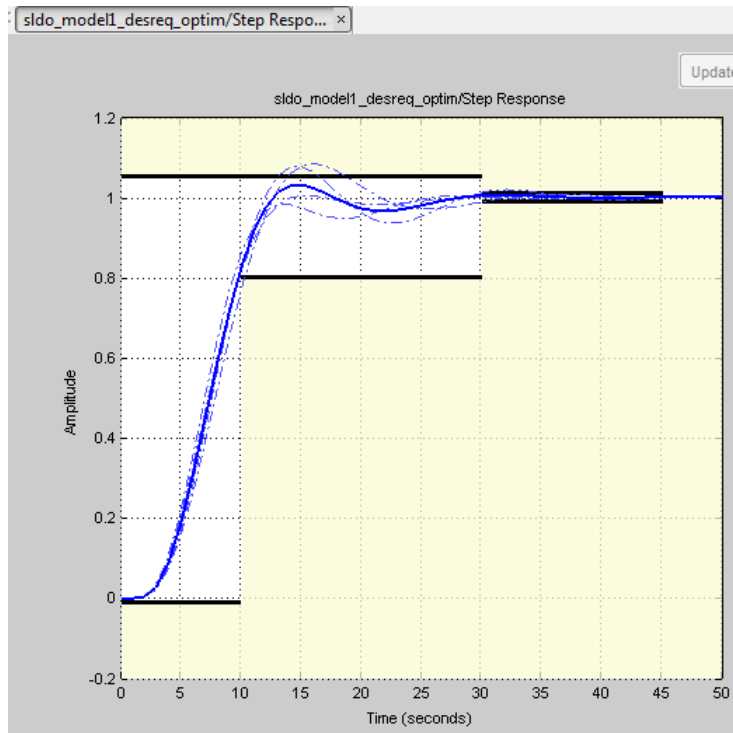
- The solid curve corresponds to the model response computed using the optimized parameters and nominal values of the uncertain parameter.

- The four dashed curves correspond to the model response with the minimum and maximum values of the uncertain parameters.

  The dashed plot lines show that the response during the period of 10 to 20 seconds violates the design requirements.

4 Optimize the parameters for model robustness. Click **Optimize**.
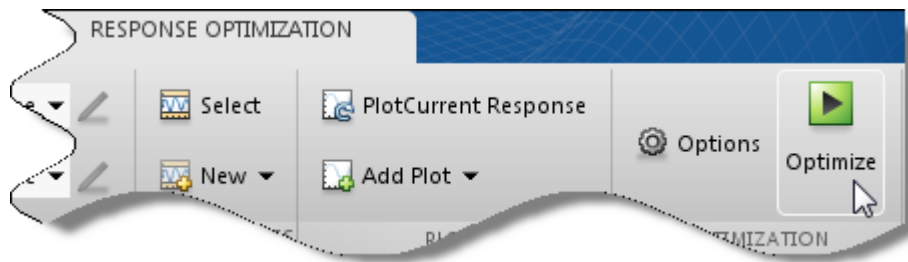
The Optimization Progress window opens which displays the optimization iterations.

After the optimization completes, the message `Optimization converged` indicates that the final model response computed by varying the uncertain parameters meets the specified design requirements.
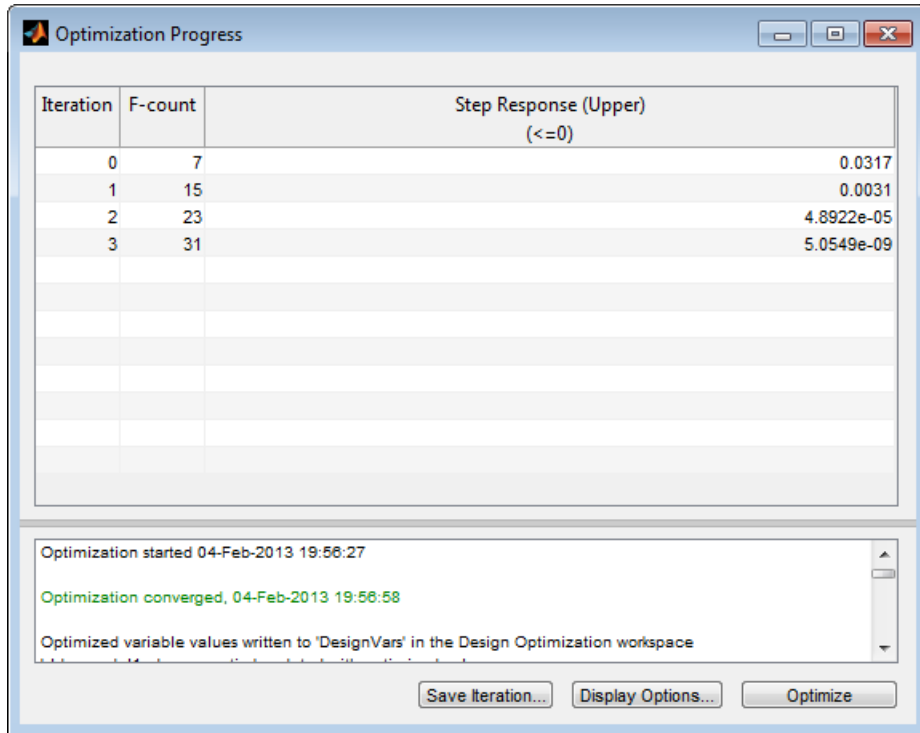
5 Examine the responses.

Tip To view only the final responses of the model, right-click the white area in the plot and uncheck **Responses > Show Iteration Responses**.
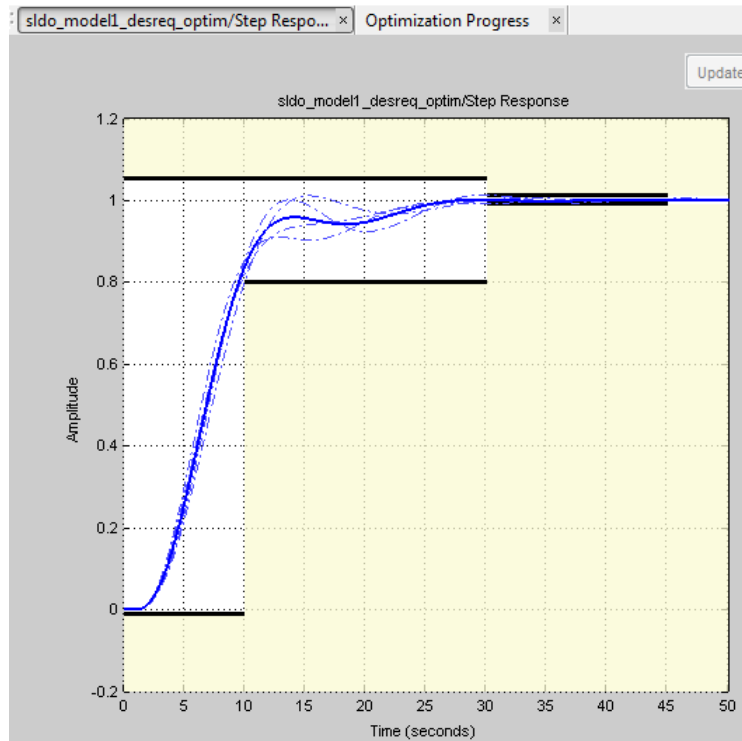
The final responses appear as the thick solid and dashed curves. The nominal and uncertain responses with parameter variations now meet the design requirements.

More About

- "What Is Robustness?" on page 3-201
- "Sampling Methods for Uncertain Parameters" on page 3-202

Related Examples

- "Design Optimization with Uncertain Variables (Code)" on page 3-171

Accelerating Model Simulations During Optimization

In this section...
"About Accelerating Optimization" on page 3-213
"Limitations" on page 3-213
"Setting Accelerator Mode for Response Optimization" on page 3-213

About Accelerating Optimization

You can accelerate the response optimization computations by changing the simulation mode of your Simulink model. Simulink Design Optimization software supports `Normal` and `Accelerator` simulation modes. For more information about these modes, see "How Acceleration Modes Work" in the Simulink documentation.

The default simulation mode is `Normal`. In this mode, Simulink uses interpreted code, rather than compiled C code during simulations.

In the `Accelerator` mode, Simulink Design Optimization software runs simulations during optimization with compiled C code. Using compiled C code speeds up the simulations and reduces the time to optimize the model response signals.

Limitations

You cannot use the `Accelerator` mode if your model contains algebraic loops. If the model contains MATLAB function blocks, you must either remove them or replace them with Fcn blocks.

If the model structure changes during optimization, the model is compiled to regenerate the C code for each iteration. Using the `Accelerator` mode increases the computation time. To learn more about code regeneration, see "Code Regeneration in Accelerated Models" in the Simulink documentation.

Setting Accelerator Mode for Response Optimization

To set the simulation mode to `Accelerator`, open the Simulink model window and perform one of the following actions:

- Select under **Simulation** > **Mode** > **Accelerator**.

- Choose `Accelerator` from the drop-down list as shown in the next figure.
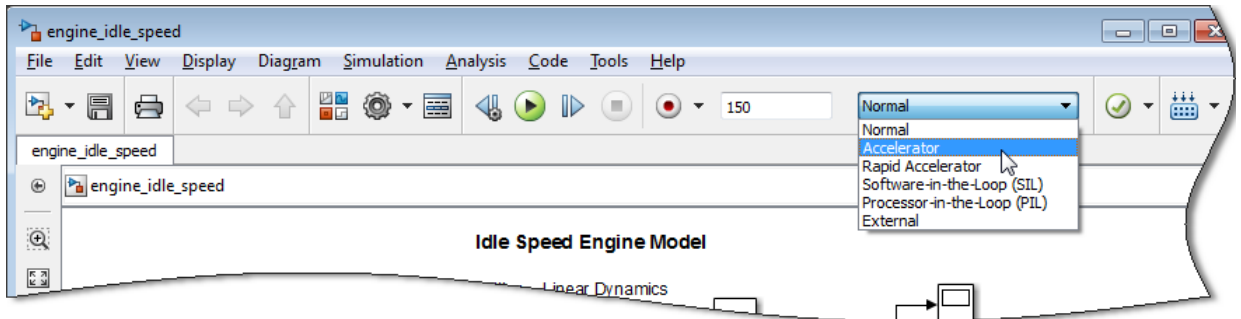
Tip To obtain the maximum performance from the `Accelerator` mode, close all Scope blocks in your model.

Speedup Response Optimization Using Parallel Computing

When to Use Parallel Computing for Response Optimization

You can use Simulink Design Optimization software with Parallel Computing Toolbox software to speed up the response optimization of a Simulink model. Using parallel computing may reduce model optimization time in the following cases:

- The model contains a large number of tuned parameters, and the `Gradient descent` method is selected for optimization.
- The `Pattern search` method is selected for optimization.
- The model contains a large number of uncertain parameters and uncertain parameter values.
- The model is complex and takes a long time to simulate.

When you use parallel computing, the software distributes independent simulations to run them in parallel on multiple MATLAB sessions, also known as *workers*. Distributing the simulations significantly reduces the optimization time because the time required to simulate the model dominates the total optimization time.

For information on how the software distributes the simulations and the expected speedup, see "How Parallel Computing Speeds Up Optimization" on page 3-215.

For information on configuring your system and using parallel computing, see "How to Use Parallel Computing for Response Optimization" on page 3-219.

How Parallel Computing Speeds Up Optimization

You can enable parallel computing with the `Gradient descent` and `Pattern search` optimization methods. When you enable parallel computing, the software distributes independent simulations during optimization on multiple MATLAB sessions. The following topics describe which simulations are distributed and the potential speedup using parallel computing:

Parallel Computing with the Gradient Descent Method

When you select `Gradient descent` as the optimization method, the model is simulated during the following computations:

- Constraint and objective value computation — One simulation per iteration
- Constraint and objective gradient computations — Two simulations for every tuned parameter per iteration
- Line search computations — Multiple simulations per iteration

The total time, T_{total}, taken per iteration to perform these simulations is given by the following equation:

$$T_{total} = T + (N_p \times 2 \times T) + (N_{ls} \times T) = T \times (1 + (2 \times N_p) + N_{ls})$$

where T is the time taken to simulate the model and is assumed to be equal for all simulations, N_p is the number of tuned parameters, and N_{ls} is the number of line searches. N_{ls} is difficult to estimate and you generally assume it to be equal to one, two, or three.

When you use parallel computing, the software distributes the simulations required for constraint and objective gradient computations. The simulation time taken per iteration when the gradient computations are performed in parallel, T_{totalP}, is approximately given by the following equation:

$$T_{totalP} = T + \left(ceil\left(\frac{N_p}{N_w}\right) \times 2 \times T\right) + (N_{ls} \times T) = T \times \left(1 + 2 \times ceil\left(\frac{N_p}{N_w}\right) + N_{ls}\right)$$

where N_w is the number of MATLAB workers.

Note: The equation does not include the time overheads associated with configuring the system for parallel computing and loading Simulink software on the remote MATLAB workers.

The expected speedup for the total optimization time is given by the following equation:

$$\frac{T_{totalP}}{T_{total}} = \frac{1 + 2 \times ceil\left(\frac{Np}{Nw}\right) + Nls}{1 + (2 \times Np) + Nls}$$

For example, for a model with N_p=3, N_w=4, and N_{1s}=3, the expected speedup equals

$$\frac{1 + 2 \times ceil\left(\frac{3}{4}\right) + 3}{1 + (2 \times 3) + 3} = 0.6 \ .$$

For an example of the performance improvement achieved with the Gradient descent method, see Improving Optimization Performance Using Parallel Computing.

Parallel Computing with the Pattern Search Method

The Pattern search optimization method uses search and poll sets to create and compute a set of candidate solutions at each optimization iteration.

The total time, T_{total}, taken per iteration to perform these simulations, is given by the following equation:

$$T_{total} = (T \times Np \times Nss) + (T \times Np \times Nps) = T \times Np \times (Nss + Nps)$$

where T is the time taken to simulate the model and is assumed to be equal for all simulations, Np is the number of tuned parameters, Nss is a factor for the search set size, and Nps is a factor for the poll set size. Nss and Nps are typically proportional to Np.

When you use parallel computing, Simulink Design Optimization software distributes the simulations required for the search and poll set computations, which are evaluated in separate parfor loops. The simulation time taken per iteration when the search and poll sets are computed in parallel, T_{totalP}, is given by the following equation:

$$T_{totalP} = (T \times ceil(Np \times \frac{Nss}{Nw})) + (T \times ceil(Np \times \frac{Nps}{Nw}))$$
$$= T \times (ceil(Np \times \frac{Nss}{Nw}) + ceil(Np \times \frac{Nps}{Nw}))$$

where Nw is the number of MATLAB workers.

Note: The equation does not include the time overheads associated with configuring the system for parallel computing and loading Simulink software on the remote MATLAB workers.

The expected speed up for the total optimization time is given by the following equation:

$$\frac{T_{totalP}}{T_{total}} = \frac{ceil(Np \times \frac{Nss}{Nw}) + ceil(Np \times \frac{Nps}{Nw})}{Np \times (Nss + Nps)}$$

For example, for a model with N_p=3, N_w=4, N_{ss}=15, and N_{ps}=2, the expected speedup

equals $\dfrac{ceil(3 \times \frac{15}{4}) + ceil(3 \times \frac{2}{4})}{3 \times (15 + 2)} = 0.27$.

Note: Using the `Pattern search` method with parallel computing may not speed up the optimization time. To learn more, see "Why do I not see the optimization speedup I expected using parallel computing?" on page 3-233

For an example of the performance improvement achieved with the `Pattern search` method, see Improving Optimization Performance Using Parallel Computing.

Related Examples

- "How to Use Parallel Computing for Response Optimization" on page 3-219

How to Use Parallel Computing for Response Optimization

In this section...
"Configure Your System for Parallel Computing" on page 3-219
"Model Dependencies" on page 3-219
"Optimize Design Using Parallel Computing (GUI)" on page 3-220
"Optimize Design Using Parallel Computing (Code)" on page 3-224
"Troubleshooting" on page 3-225

Configure Your System for Parallel Computing

You can speed up model optimization using parallel computing on multicore processors or multiprocessor networks. Use parallel computing with the Response Optimization tool and `sdo.optimize` to optimize using the `fmincon`, `lsqonlin`, and `patternsearch` methods. Parallel computing is not supported for the `fminsearch` (**Simplex search**) method.

When you optimize model parameters using parallel computing, the software uses the available parallel pool. If none is available, and you select **Automatically create a parallel pool** in your Parallel Computing Toolbox preferences, the software starts a parallel pool using the settings in those preferences. To open a parallel pool that uses a specific cluster profile, use:

```
parpool(MyProfile);
```

`MyProfile` is the name of a cluster profile.

For information regarding creating a cluster profile, see "Create and Modify Cluster Profiles" in the Parallel Computing Toolbox documentation.

Model Dependencies

Model dependencies are any referenced models, data such as model variables, S-functions, and additional files necessary to run the model. Before starting the optimization, verify that the model dependencies are complete. Otherwise, you may get unexpected results.

Making File Dependencies Accessible to Remote Workers

When you use parallel computing, the Simulink Design Optimization software helps you identify model dependencies. To do so, the software uses the Simulink Manifest Tools. The dependency analysis may not find all the files required by your model. To learn more, see "Scope of Dependency Analysis" in the Simulink documentation. If your model has dependencies that are undetected or inaccessible by the parallel pool workers, then add them to the list of model dependencies.

The dependencies 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.
- Path dependencies: the paths to the model dependencies are specified to the parallel pool workers. If you are working in a multi-platform scenario, ensure that the paths are compatible across platforms.

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.

For more information, see:

- "Optimize Design Using Parallel Computing (GUI)" on page 3-220
- "Optimize Design Using Parallel Computing (Code)" on page 3-224

Optimize Design Using Parallel Computing (GUI)

To optimize a model response using parallel computing in the Response Optimization tool:

1   Ensure that the software can access parallel pool workers that use the appropriate cluster profile.

    For more information, see "Configure Your System for Parallel Computing" on page 3-219.

2   Open the Response Optimization tool for the Simulink model.

3   Configure the design variables, design requirements, and, optionally, optimization settings.

For more information, see "Specify Design Variables" on page 3-61, "Specify Time-Domain Design Requirements" on page 3-23, "Specify Frequency-Domain Design Requirements" on page 3-39, and "Optimization Options" on page 3-73.

4

On the **Response Optimization** tab, click ⚙**Options** to open the **Response Optimization Options** dialog box.

5 Select the **Parallel Options** tab.

6 Select the **Use the parallel pool during optimization** check box.

This option checks for dependencies in your Simulink model. The file dependencies are displayed in the **Model file dependencies** list box, and corresponding path to the files in **Model path dependencies**. The files listed in **Model file dependencies** are copied to the remote workers.

Note: The automatic dependencies check may not detect all the dependencies in your model.

For more information, see "Model Dependencies" on page 3-219. In this case, add the undetected dependencies manually.

7 Add any file dependencies that the automatic check does not detect.

Specify the files in the **Model file dependencies** list box separated by semicolons or on separate lines.

Alternatively, click **Add file dependency** to open a dialog box, and select the file to add.

Note: If you do not want to copy the files to the remote workers, delete all entries in the **Model file dependencies** list box. Populate the **Model path dependencies** list box by clicking the **Sync path dependencies from model**, and add any undetected path dependencies. In addition, in the list box, update the paths on local drives to make them accessible to remote workers. For example, change `C:\` to `\\\ \hostname\\C$\\`.

---

8   If you modify the Simulink model, resync the dependencies to ensure that any new dependencies are detected. Click **Sync file dependencies from model** in the **Parallel Options** tab to rerun the automatic dependency check for your model.

This action updates the **Model file dependencies** list box with any new file dependency found in the model.

9   Click **OK**.

10 In the Response Optimization tool, click **Optimize** to optimize the model response using parallel computing.

For information on troubleshooting problems related to optimization using parallel computing, see "Troubleshooting" on page 3-225.

Optimize Design Using Parallel Computing (Code)

To optimize a model response using parallel computing at the command line:

1 Ensure that the software can access parallel pool workers that use the appropriate cluster profile.

For more information, see "Configure Your System for Parallel Computing" on page 3-219.

2 Open the model.

3 Specify design requirements and design variables. For example see, "Design Optimization to Meet Step Response Requirements (Code)".

4 Enable parallel computing using an optimization option set, `opt`.

```
opt = sdo.OptimizeOptions;
opt.UseParallel = 'always';
```

5 Find the model dependencies.

```
[dirs, files] = sdo.getModelDependencies(modelname)
```

Note: `sdo.getModelDependencies` may not detect all the dependencies in your model. For more information, see "Model Dependencies" on page 3-219. In this case, add the undetected dependencies manually.

---

6 Modify `files` to include any file dependencies that `sdo.getModelDependencies` does not detect.

```
files = vertcat(files,'C:\matlab\work\filename.m')
```

Note: If you do not want to copy the files to the remote workers, use the path dependencies. Add any undetected path dependencies to `dirs` and update the paths on local drives to make them accessible to remote workers. See `sdo.getModelDependencies` for more details.

---

7 Add the file dependencies for optimization.

```
opt.ParallelFileDependencies = files;
```

8   Run the optimization.

```
[pOpt,opt_info] = sdo.optimize(opt_fcn,param,opt);
```

For information on troubleshooting problems related to optimization using parallel computing, see "Troubleshooting" on page 3-225.

Troubleshooting

- "Why Are the Optimization Results With and Without Using Parallel Computing Different?" on page 3-225
- "Why Don't I See the Optimization Speed up I Expected Using Parallel Computing?" on page 3-225
- "Why Doesn't the Optimization Using Parallel Computing Make Any Progress?" on page 3-226
- "Why Doesn't the Optimization Using Parallel Computing Stop When I Click the Stop Optimization Button?" on page 3-226

Why Are the Optimization Results With and Without Using Parallel Computing Different?

- Different numerical precision on the client and worker machines can produce marginally different simulation results. Thus, the optimization method can take a different solution path and produce a different result.

- When you use parallel computing with the `Pattern search` method, the search is more comprehensive and can result in a different solution.

  To learn more, see "Parallel Computing with the Pattern Search Method" on page 3-217.

Why Don't I See the Optimization Speed up I Expected Using Parallel Computing?

- When you optimize a model that does not have a large number of parameters or does not take long to simulate, you might not see a speedup in the optimization time. In such cases, the overhead associated with creating and distributing the parallel tasks outweighs the benefits of running the optimization in parallel.

- Using the `Pattern search` method with parallel computing might not speed up the optimization time. Without parallel computing, the method stops the search at each iteration when it finds a solution better than the current solution. The candidate solution search is more comprehensive when you use parallel computing. Although

the number of iterations might be larger, the optimization without using parallel computing might be faster.

To learn more about the expected speedup, see "Parallel Computing with the Pattern Search Method".

Why Doesn't the Optimization Using Parallel Computing Make Any Progress?

In some cases, the gradient computations on the remote worker machines may silently error out when you use parallel computing. In such cases, the Optimization Progress window shows that the `f(x)` and `max constraint` values do not change, and the optimization terminates after two iterations with the message `Unable to satisfy constraints`. To troubleshoot the problem:

1 Run the optimization for a few iterations without parallel computing to see if the optimization progresses.
2 Check whether the remote workers have access to all model dependencies. Model dependencies include data variables and files required by the model to run.

   To learn more, see "Model Dependencies".

Why Doesn't the Optimization Using Parallel Computing Stop When I Click the Stop Optimization Button?

When you use parallel computing with the `Pattern search` method, the software must wait until the current optimization iteration completes before it notifies the workers to stop. The optimization does not terminate immediately when you click **Stop**, and, instead, appears to continue running.

See Also
sdo.OptimizeOptions | parpool | sdo.getModelDependencies | sdo.optimize

Related Examples
- "Optimizing Time-Domain Response of Simulink® Models Using Parallel Computing" on page 3-240

More About
- "Speedup Response Optimization Using Parallel Computing" on page 3-215

Optimization Does Not Make Progress

In this section...
"Should I worry about the scale of my responses and how constraints and design requirements are discretized?" on page 3-227
"Why don't the responses and parameter values change at all?" on page 3-227
"Why does the optimization stall?" on page 3-227

Should I worry about the scale of my responses and how constraints and design requirements are discretized?

No. Simulink Design Optimization software automatically normalizes constraints, design requirement and response data.

Why don't the responses and parameter values change at all?

The optimization problem you formulated might be nonsmooth. This means that small parameter changes have no effect on the amount by which response signals satisfy or violate the constraints and only large changes will make a difference. Try switching to a search-based method such as simplex search or pattern search. Alternatively, look for initial guesses outside of the dead zone where parameter changes have no effect. If you are optimizing the response of a Simulink model, you could also try removing nonlinear blocks such as Quantizer or Dead Zone.

Why does the optimization stall?

When optimizing a Simulink model, certain parameter combinations can make the simulation stall for models with strong nonlinearities or frequent mode switching. In these cases, the ODE solvers take smaller and smaller step sizes. Stalling can also occur when the model's ODEs become too stiff for some parameter combinations. A symptom of this behavior is when the Simulink model status is `Running` and clicking the **Stop** button fails to interrupt the optimization. When this happens, you can try one of the following solutions:

- Switch to a different ODE solver, especially one of the stiff solvers.
- Specify a minimum step size.

- Disable zero crossing detection if chattering is occurring.
- Tighten the lower and upper bounds on parameters that cause simulation difficulties. In particular, eliminate regions of the parameter space where some model assumptions are invalid and the model behavior can become erratic.

Optimization Convergence

In this section...
"What to do if the optimization does not get close to an acceptable solution?" on page 3-229
"Why does the optimization terminate before exceeding the maximum number of iterations, with a solution that does not satisfy all the constraints or design requirements?" on page 3-230
"What to do if the optimization takes a long time to converge even though it is close to a solution?" on page 3-230
"What to do if the response becomes unstable and does not recover?" on page 3-231

What to do if the optimization does not get close to an acceptable solution?

- If you are using pattern search, check that you have specified appropriate maximum and minimum values for all your tuned parameters or compensator elements. The pattern search method looks inside these bounds for a solution. When they are set to their default values of `Inf` and `-Inf`, the method searches within ±100% of the initial values of the parameters. In some cases this region is not large enough and changing the maximum and minimum values can expand the search region.

- Your optimization problem might have local minima. Consider running one of the search-based methods first to get closer to an acceptable solution.

- Reduce the number of tuned parameters and compensator elements by removing from the design variables or from the **Compensators** pane (when using a SISO Design Task) those parameters that you know only mildly influence the optimized responses. After you identify reasonable values for the key parameters, add the fixed parameters back to the tunable list and restart the optimization using these reasonable values as initial guesses.

- The software may have encountered errors during the optimization. Review the errors to determine if you can make changes to improve the optimization results. Changes may require modifications to the model, requirements, or optimization settings.

  - In the Response Optimization tool, the software creates a structure named `EvalErrors` in the **Data** area when the optimization completes with errors. Export this structure to the MATLAB workspace and examine its contents at the

command line. `EvalErrors` has two fields, `Errors` and `DesignVars`, containing the errors encountered during optimization and the corresponding design variable values. To reproduce a specific error, use `sdo.setValueInModel` to run the model using the design variables that correspond to the error.

- At the command line, the second output of `sdo.optimize`, `opt_info`, is a structure that provides information regarding the optimization. `opt_info.exitflag` identifies the reason the optimization terminated. For more information regarding exit flags, see "Exit Flags and Exit Messages".

Why does the optimization terminate before exceeding the maximum number of iterations, with a solution that does not satisfy all the constraints or design requirements?

- It might not be possible to achieve your specifications. Try relaxing the constraints or design requirements that the response signals violate the most. After you find an acceptable solution to the relaxed problem, tighten some constraints again and restart the optimization.

- The optimization might have converged to a local minimum that is not a feasible solution. Restart the optimization from a different initial guess and/or use one of the search-based methods to identify another local minimum that satisfies the constraints.

What to do if the optimization takes a long time to converge even though it is close to a solution?

- In a Response Optimization tool, click **Stop** to interrupt the optimization when you think the current optimized response signals are acceptable.

  When you use a SISO Design Task, click **Stop Optimization** in the **Optimization** panel of the **Response Optimization** node in the Control and Estimation Tools Manager, when you think the current optimized response signals are acceptable.

- If you use the gradient descent method, try restarting the optimization. Restarting resets the Hessian estimate and might speed up convergence.

- Increase the convergence tolerances in the Optimization Options dialog to force earlier termination.

- Relax some of the constraints or design requirements to increase the size of the feasibility region.

What to do if the response becomes unstable and does not recover?

While the optimization formulation has explicit safeguards against unstable or divergent response signals, the optimization can sometimes venture into an unstable region where simulation results become erratic and gradient methods fail to find a way back to the stable region. In these cases, you can try one of the following solutions:

- Add or tighten the lower and upper bounds on compensator element and parameter values. Instability often occurs when you allow some parameter values to become too large.

- Use a search-based method to find parameter values that stabilize the response signals and then start the gradient-based method using these initial values.

- When optimizing responses in a SISO Design Task, you can try adding additional design requirements that achieve the same or similar goal. For example, in addition to a settling time design requirement on a step response plot, you could add a settling time design requirement on a root-locus plot that restricts the location of the real parts of the poles. By adding overlapping design requirements in this way, you can force the optimization to meet the requirements.

Optimization Speed and Parallel Computing

How can I speed up the optimization?

- The optimization time is dominated by the time it takes to simulate the model. When optimizing a Simulink model, you can enable the Accelerator mode using **Simulation** > **Mode** > **Accelerator** in the Simulink Editor, to dramatically reduce the optimization time.

  Note: The Rapid Accelerator mode in Simulink software is not supported for speeding up the optimization. For more information, see "Accelerating Model Simulations During Optimization".

- The choice of ODE solver can also significantly affect the overall optimization time. Use a stiff solver when the simulation takes many small steps, and use a fixed-step solver when such solvers yield accurate enough simulations for your model. (These solvers must be accurate in the entire parameter search space.)

- Reduce the number of tuned compensator elements or parameters and constrain their range to narrow the search space.

- When specifying parameter uncertainty (not available when optimizing responses in a SISO Design Task), keep the number of sample values small since the number of simulations grows exponentially with the number of samples. For example, a grid of 3 parameters with 10 sample values for each parameter requires 10^3=1000 simulations per iteration.

Why are the optimization results with and without using parallel computing different?

- Different numerical precision on the client and worker machines can produce marginally different simulation results. Thus, the optimization method can take a different solution path and produce a different result.

- When you use parallel computing with the `Pattern search` method, the search is more comprehensive and can result in a different solution.

  To learn more, see "Parallel Computing with the Pattern Search Method" on page 3-217.

Why do I not see the optimization speedup I expected using parallel computing?

- When you optimize a model that does not have a large number of parameters or does not take long to simulate, you might not see a speedup in the optimization time. In such cases, the overhead associated with creating and distributing the parallel tasks outweighs the benefits of running the optimization in parallel.

- Using the `Pattern search` method with parallel computing might not speed up the optimization time. Without parallel computing, the method stops the search at each iteration when it finds a solution better than the current solution. The candidate solution search is more comprehensive when you use parallel computing. Although the number of iterations might be larger, the optimization without using parallel computing might be faster.

  To learn more about the expected speedup, see "Parallel Computing with the Pattern Search Method".

Why does the optimization using parallel computing not make any progress?

In some cases, the gradient computations on the remote worker machines may silently error out when you use parallel computing. In such cases, the Optimization Progress window shows that the `f(x)` and `max constraint` values do not change, and the optimization terminates after two iterations with the message `Unable to satisfy constraints`. To troubleshoot the problem:

1 Run the optimization for a few iterations without parallel computing to see if the optimization progresses.

2 Check whether the remote workers have access to all model dependencies. Model dependencies include data variables and files required by the model to run.

To learn more, see "Model Dependencies".

Why does the optimization using parallel computing not stop when I click the Stop optimization button?

When you use parallel computing with the `Pattern search` method, the software must wait until the current optimization iteration completes before it notifies the workers to stop. The optimization does not terminate immediately when you click **Stop**, and, instead, appears to continue running.

Undesirable Parameter Values

In this section...
"What to do if the optimization drives the tuned compensator elements and parameters to undesirable values?" on page 3-235
"What to do if the optimization violates bounds on parameter values?" on page 3-235

What to do if the optimization drives the tuned compensator elements and parameters to undesirable values?

- When a tuned compensator element or parameter is positive, or when its value is physically constrained to a given range, enter the lower and upper bounds (**Minimum** and **Maximum**) in one of the following:

    - Dialog box to select design variables (in Response optimization tool)
    - **Compensators** pane (in a SISO Design Task)

    This information helps guide the optimization method towards a reasonable solution.
- Specify initial guesses that are within the range of desirable values.
- In the **Compensators** pane in a SISO Design Task, verify that no integrators/ differentiators are selected for optimization. Optimizing the pole/zero location of integrators/differentiators can result in drastic changes in the system gain and lead to undesirable values.

What to do if the optimization violates bounds on parameter values?

The `Gradient descent` optimization method fmincon violates the parameter bounds when it cannot simultaneously satisfy the signal constraints and the bounds. When this happens, try one of the following:

- Specify a different value for the parameter bound and restart the optimization. A guideline is to adjust the bound by 1% of the typical value.

    For example, for a parameter with a typical value of `1` and lower bound of `0`, change the lower bound to `0.01`.
- Relax the signal constraints and restart the optimization. This approach results in a different solution path for the `Gradient descent` method.

- Restart the optimization immediately after it terminates by clicking **Optimize** in the Response Optimization tool. This approach uses the previous optimization results as the starting point for the next optimization cycle to refine the results.

- Use the following two-step approach to perform the optimization:

  1 Run an initial optimization to satisfy the signal constraints.

  For example, run the optimization using the `Simplex search` method. This method satisfies the signal constraints but does not support the bounds on parameter values. The results obtained using this method provide the starting point for the optimization performed in the next step. To learn more about this method, see the fminsearch function reference page in the Optimization Toolbox documentation.

  2 Reconfigure the optimization by selecting a different optimization method to satisfy both the signal constraints and the parameter bounds.

  For example, change the optimization method to `Gradient descent` and run the optimization again.

Tip If Global Optimization Toolbox software is installed, you can select the `Pattern search` optimization method to optimize the model response.

Reverting to Initial Parameter Values

How do I quit an optimization and revert to my initial parameter values?

- Before running an optimization, do one of the following:

  - In the Response Optimization tool, click Options. Uncheck **Update model at end of optimization** in the **General Options** tab.

  - In the Response Optimization tool, click Options. Select **Save optimized variable values as new design variable set** in the **General Options** tab.

  - Make a copy of the design variable set in the **Data** area.

  If you want to revert to the initial parameter values after the optimization terminates or you stop the optimization by clicking **Stop**, select the design variable that contains the initial values in the **Design Variable Set** drop-down list and click adjacent to **Design variables Set**. Select the design variables in the dialog box and click **Update model variable values** to revert the model parameters to their original values.

- When using a SISO Design Task, the **Start Optimization** button becomes a **Stop Optimization** button after the optimization has begun. To quit the optimization, click the **Stop Optimization** button. To revert to the initial parameter values, select **Edit** > **Undo Optimize compensators** from the menu in the SISO Design Tool window.

Manage Response Optimization Tool Session

In this section...
"Save a Session" on page 3-238
"Load a Session" on page 3-238

Save a Session

Saving a session lets you reuse your settings and optimization results later. These settings include design requirements, design and uncertain variables, plots and optimization settings. Each Response Optimization tool session is associated with a Simulink model.

You can save the session as a MAT-file or workspace variable:

- To save the session as a MAT-file, in the **Save Session** drop-down list, click **Save to file** in the **Response Optimization** tab. A window opens where you specify the MAT-file name.
- To save the session as a model or MATLAB workspace variable, select **Save to model workspace** or **Save to MATLAB workspace** in the **Save Session** drop-down list.
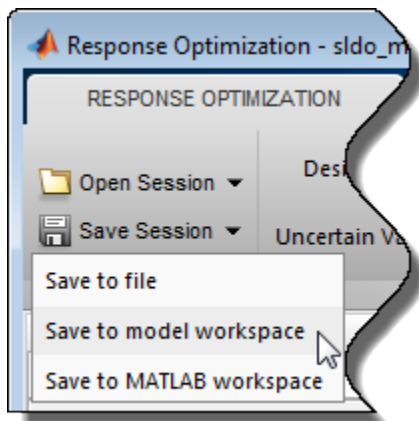
Load a Session

You can load a saved MAT-file or workspace session:

1 Open a Response Optimization tool for the model.

2 To load a MAT-file, in the **Response Optimization** tab click on the **Open Session** drop-down list, and select **Open from file** . A window opens where you select the MAT-file to load.

To load a workspace variable, select **Open from model workspace** or **Open from MATLAB workspace** in the **Open Session** drop-down list.
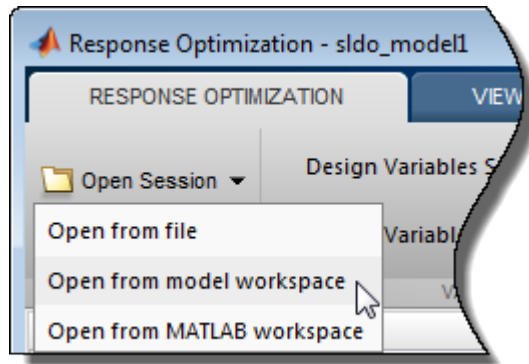
Optimizing Time-Domain Response of Simulink® Models Using Parallel Computing

This example shows how to use parallel computing to optimize the time-domain response of a Simulink® model. You use Simulink® Design Optimization™ and Parallel Computing Toolbox™ to tune the gains of a discrete PI controller of a boiler to meet the design requirements. The example also shows how the software automatically handles model file dependencies.

This example requires Parallel Computing Toolbox™.

Opening the Model

The Simulink model consists of a boiler model and a discrete PI controller. When using parallel computing, Simulink Design Optimization performs a model dependency check, which recognizes the boiler model library as an installed library.

In order to illustrate how model dependencies are handled when using parallel computing, we copy the boiler model and library block to a temporary folder before opening the model.

```
pathToLib = boilerpressure_setup;   %Copies boiler model and library to a temporary fol
addpath(pathToLib);
open_system('boilerpressure_demo')
```

Boiler control

Copyright 2004-2013 The MathWorks, Inc.

Two state Boiler model with constant downstream exit pressure. States are steam temperature and steam volume. Discrete PI controller to maintain drum pressure at nominal condition.

Click on the Optimize button in the GUI to optimize the Kp, Ki gains of the PI Pressure regulator block.
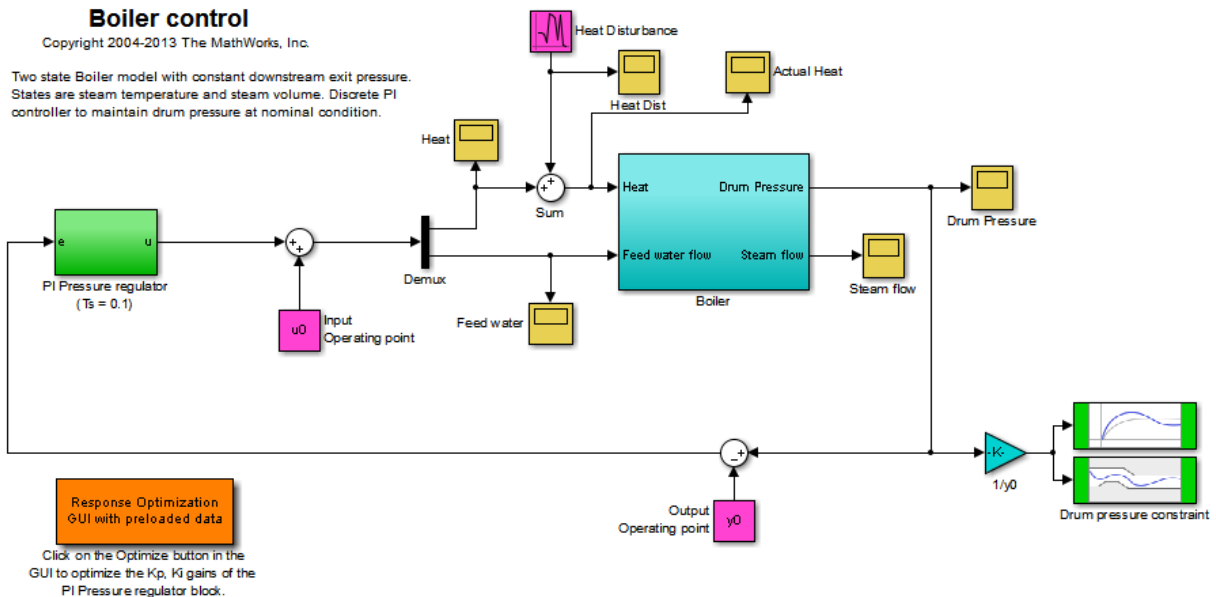
Design Requirements

The boiler pressure is regulated by a discrete PI controller. The design requirement for the controller is to limit the pressure variation of the boiler within +-%5 of the nominal pressure.

The initial controller has fairly good regulation characteristics but in the presence of additional heat disturbances, modeled by the Heat Disturbance block, we want to tune the controller performance to provide tighter pressure regulation.

Double-click the 'Response Optimization GUI with preloaded data' block in the Simulink model to open a pre-configured Response Optimization Tool. The Response Optimization Tool is configured with:

1. Upper and lower bounds representing a +-5% allowable range on the drum pressure

2. A reference tracking objective to minimize the deviation of the drum pressure from nominal

3. The PI controller gains, Kp and Ki, are selected for tuning

Click **Plot Model Response** to display the drum pressure variations with the initial controller.
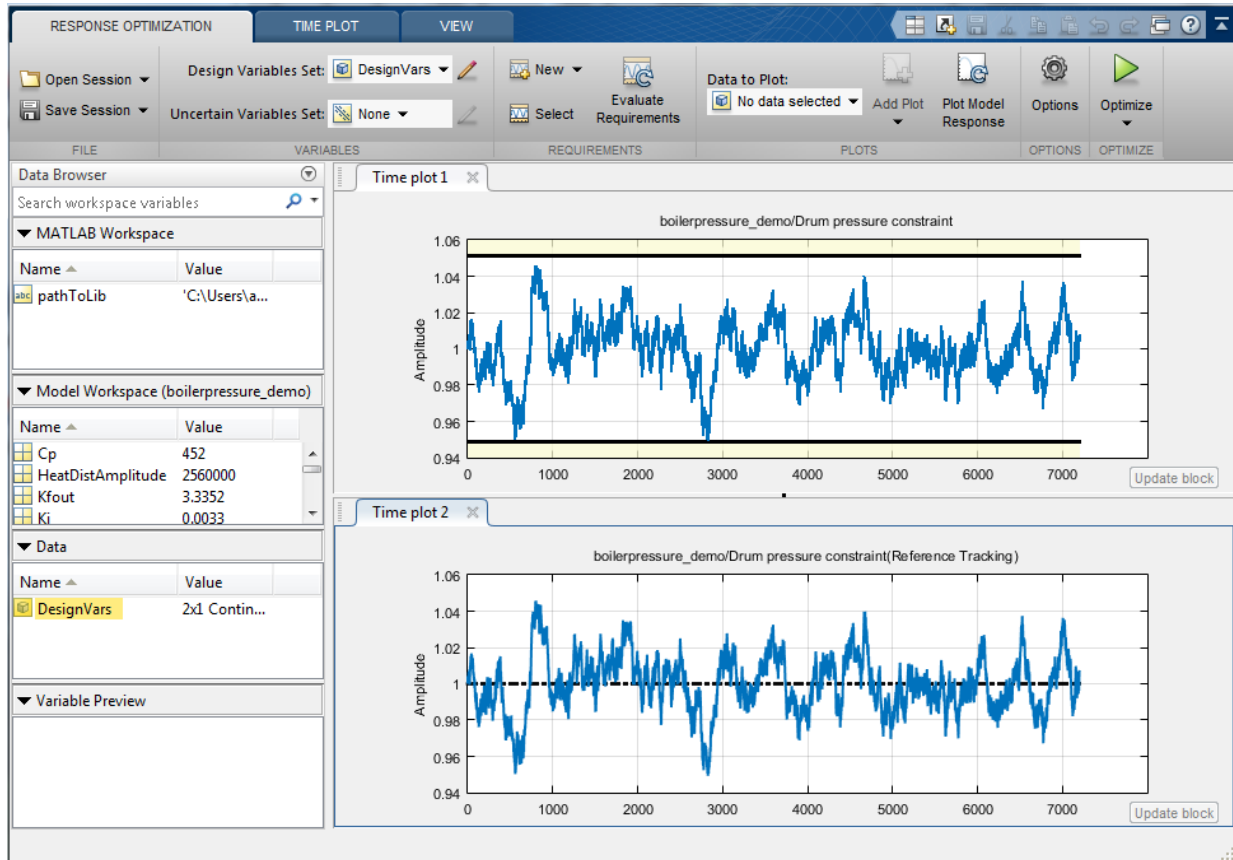
Configuring and Running the Optimization in the GUI Using Parallel Computing

When computing the model response with the initial controller, this complex model took a long time to simulate. Using parallel computing can reduce the optimization time by simulating the model in parallel. For more information on parallel computing and optimization performance see the tutorial "Improving Optimization Performance Using Parallel Computing".

To configure the optimization problem to use parallel computing click **Options** in the Response Optimization Tool and select the **Parallel Options** tab. Select the "Use the

parallel pool during optimization" option. This triggers an automated search for any model dependencies. In this example, the Steam table library (`libsteam.slx`) is found as a model dependency (in addition to the `boilerpressure_demo` model itself), and is displayed in the Model file dependencies list box.
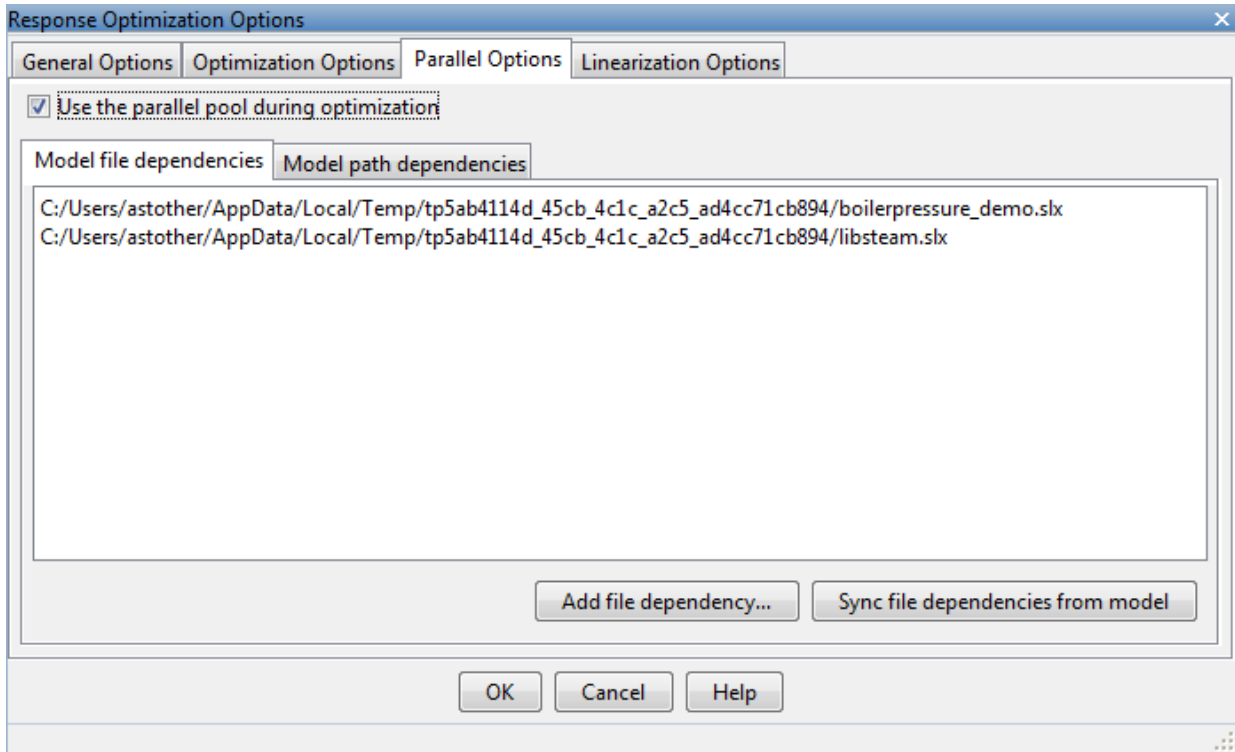
Clicking OK configures the optimization to use parallel computing.

To run the optimization click the **Optimize** button. A progress window opens displaying optimization progress and the plots update to show the optimized response.
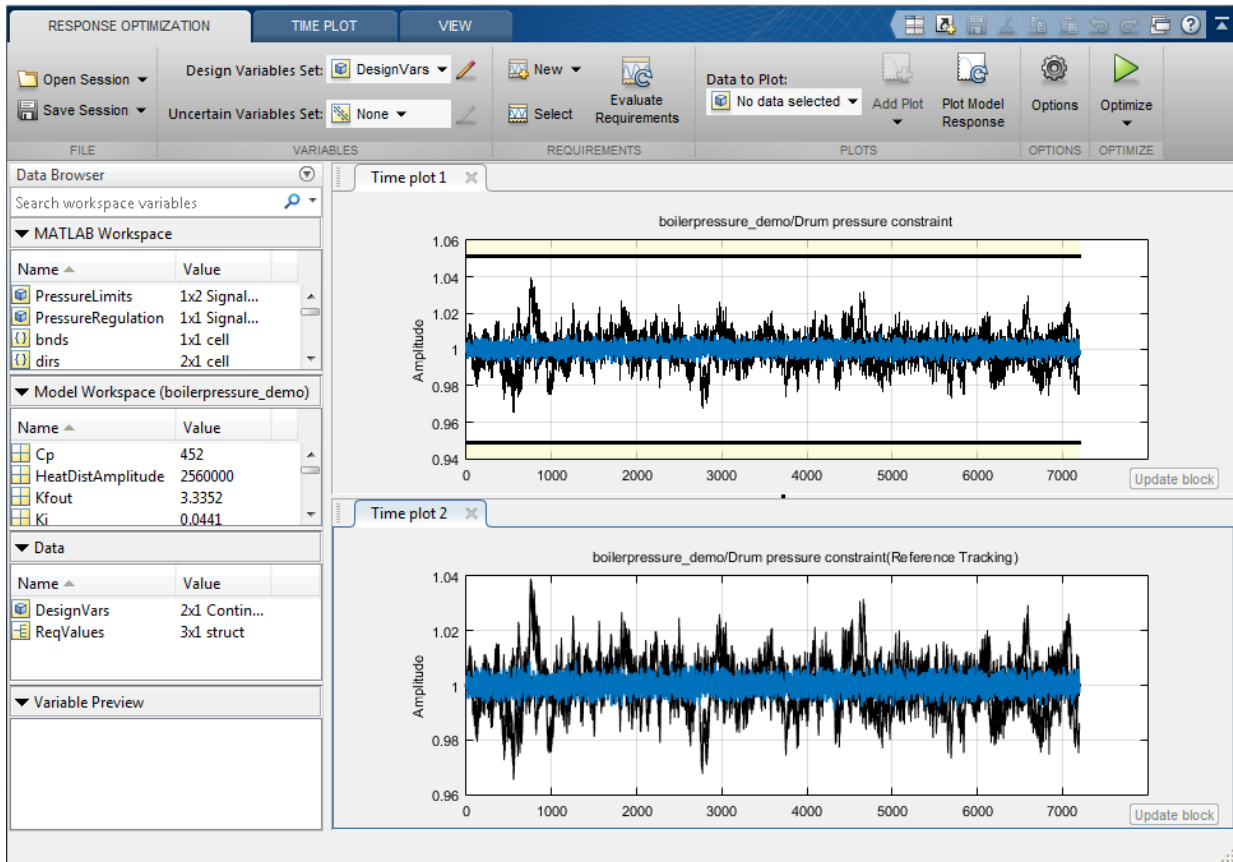
The final response shows that the optimized regulator tracks the reference pressure much more closely and the drum pressure constraints are satisfied.

Configuring and Running the Optimization at the Command Line

You can also use the command line functions to configure the optimization to use parallel computing and run the optimization.

Select the model variables for optimization and set lower limits

```
p = sdo.getParameterFromModel('boilerpressure_demo',{'Kp','Ki'});
p(1).Minimum = 0.001;
p(2).Minimum = 0.001;
```

Select the model signal to bound and create a simulator to simulate the model.

```
nPressure = Simulink.SimulationData.SignalLoggingInfo;
nPressure.BlockPath              = 'boilerpressure_demo/1//yO';
nPressure.OutputPortIndex        = 1;
nPressure.LoggingInfo.NameMode   = 1;
nPressure.LoggingInfo.LoggingName = 'nPressure';

simulator = sdo.SimulationTest('boilerpressure_demo');
simulator.LoggingInfo.Signals = nPressure;
```

Get the optimization requirements defined by the check blocks in the model so that we can use them in the optimization problem.

```
bnds = getbounds('boilerpressure_demo/Drum pressure constraint');
PressureLimits = [bnds{:}];
bnds = getbounds('boilerpressure_demo/Drum pressure constraint(Reference Tracking)');
PressureRegulation = [bnds{:}];
requirements = struct(...
    'PressureLimits', PressureLimits, ...
    'PressureRegulation', PressureRegulation);
```

Define the function called during optimization. Notice that the function uses the simulator and requirements defined earlier to evaluate the design.

```
evalDesign = @(p) boilerpressure_design(p,simulator,requirements);
type boilerpressure_design


function design = boilerpressure_design(p,simulator,requirements)
%BOILERPRESSURE_DESIGN
%
% The boilerpressure_design function is used to evaluate a boiler
% controller design design.
%
% The |p| input argument is the vector of controller parameters.
%
% The |simulator| input argument is a sdo.SimulinkTest object used to
% simulate the |boilerpressure_demo| model and log simulation signals.
%
% The |requirements| input argument contains the design requirements used
% to evaluate the boiler controller design.
%
% The |design| return argument contains information about the design
% evaluation that can be used by the |sdo.optimize| function to optimize
```

```
% the design.
%
% see also sdo.optimize, sdoExampleCostFunction

% Copyright 2011 The MathWorks, Inc.

%% Simulate the model
%
% Use the simulator input argument to simulate the model and log model
% signals.
%
% First ensure that we simulate the model with the parameter values chosen
% by the optimizer.
%
simulator.Parameters = p;
%%
% Simulate the model and log signals.
%
simulator = sim(simulator);
%%
% Get the simulation signal log, the simulation log name is defined by the
% model |SignalLoggingName| property
%
logName = get_param('boilerpressure_demo','SignalLoggingName');
simLog = get(simulator.LoggedData,logName);

%% Evaluate the design requirements
%
% Use the requirements input argument to evaluate the design requirements
%
% Check the Pressure signal against the |PressureLimits| requirements.
%
nPressure = get(simLog,'nPressure');
c = [...
    evalRequirement(requirements.PressureLimits(1),nPressure.Values); ...
    evalRequirement(requirements.PressureLimits(2),nPressure.Values)];
%%
% Use the PressureLimits requirements as non-linear constraints for
% optimization.
design.Cleq = c(:);
%%
% Check the pressure signal against the |PressureRegulation| requirement.
%
f = evalRequirement(requirements.PressureRegulation,nPressure.Values);
```

```
%%
% Use the PressureRegulation requirement as an objective for optimization.
design.F = f;
end
```

Setup optimization options to use the parallel pool and specify the model and model files dependencies.

```
opt = sdo.OptimizeOptions;
opt.UseParallel              = 'always';
opt.OptimizedModel           = 'boilerpressure_demo';
[dirs,files]                 = sdo.getModelDependencies('boilerpressure_demo');
opt.ParallelFileDependencies = files;
```

To run the optimization using the parallel pool pass the optimization options, `opt`, to the `sdo.optimize` command.

```
>> [pOpt,info] = sdo.optimize(evalDesign,p,opt);
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Configuring parallel workers for optimization...
Parallel workers configured for optimization.

 Optimization started 18-Nov-2014 10:31:21

                               max       Step-size    First-order
 Iter F-count        f(x)   constraint                optimality
    0       5      17.5068           0
    1      10      11.6563           0        1.25         32.2
    2      15      8.32632           0        1.27         17.4
    3      20     0.694611           0        70.1       0.0391
    4      25     0.562489           0         4.3       0.0355
    5      31     0.543708           0        2.79       0.0316
    6      36     0.543708           0      0.0515       0.0316
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the selected value of the step size tolerance and constraints are
satisfied to within the selected value of the constraint tolerance.
Removing data from parallel workers...
Data removed from parallel workers.
```

Closing the Model

After the model is optimized, we remove the boiler model and library file from the temporary folder.

```
rmpath(pathToLib)
boilerpressure_cleanup(pathToLib)
bdclose('boilerpressure_demo')
```

4

Sensitivity Analysis

What Is Sensitivity Analysis?

Generally, *sensitivity analysis* is defined as the study of how uncertainty in the output of a model can be attributed to different sources of uncertainty in the model input[1]. In the context of using Simulink Design Optimization software, sensitivity analysis refers to understanding how the parameters and states (optimization *design variables*) of a Simulink model influence the optimization cost function. Examples of using sensitivity analysis include:

- Before optimization — Determine the influence of the parameters of a Simulink model on the output. Use sensitivity analysis to rank parameters in order of influence so that you can determine the most influential parameters. Optimize the model by tuning the most influential parameters or perform experiments to better characterize those parameters.
- After optimization — Test how robust the cost function is to small changes in the values of optimized parameters.

One approach to sensitivity analysis is *local sensitivity analysis*, which is derivative based (numerical or analytical). Mathematically, the sensitivity of the cost function with respect to certain parameters is equal to the partial derivative of the cost function with respect to those parameters. The term *local* refers to the fact that all derivatives are taken at a single point. For simple cost functions, this approach is efficient. However, this approach can be infeasible for complex models, where formulating the cost function (or the partial derivatives) is nontrivial. For example, models with discontinuities do not always have derivatives.

Local sensitivity analysis is a *one-at-a-time* (OAT) technique. OAT techniques analyze the effect of one parameter on the cost function at a time, keeping the other parameters fixed. They explore only a small fraction of the design space, especially when there are many parameters. Also, they do not provide insight about how the interactions between parameters influence the cost function.

Another approach to sensitivity analysis is *global sensitivity analysis*, often implemented using Monte Carlo techniques. This approach uses a representative (global) set of samples to explore the design space. Use Simulink Design Optimization software to perform global sensitivity analysis. The workflow is as follows:

1   Sample the model parameters using experimental design principles. That is, for each parameter, generate multiple values that the parameter can assume. Define the parameter sample space by specifying probability distributions for each parameter. You can also specify parameter correlations.

For information about sampling parameters, see "Sampling Parameters for Sensitivity Analysis" on page 4-4.

2 Evaluate the optimization cost function at each sample point. You can plot the cost function output for the samples to visually analyze trends.

3 (Optional) Formally analyze the relation between the cost function and the samples. Analysis methods include correlation, partial correlation (requires a Statistics and Machine Learning Toolbox™ license), and standardized regression. You can configure each analysis method to use either raw or ranked data.

For information about the analysis methods, see "Sensitivity Analysis Methods" on page 4-11.

References

[1] Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., and Tarantola, S. *Global Sensitivity Analysis. The Primer*, John Wiley and Sons, 2008.

See Also

sdo.analyze | sdo.evaluate | sdo.sample | sdo.scatterPlot

Related Examples

More About

Sampling Parameters for Sensitivity Analysis

You can perform global sensitivity analysis using Simulink Design Optimization software. You vary the value of the Simulink model parameters and states of interest in a specific range. These parameters and states are the optimization *design variables*, collectively referred to as *parameters*. Each combination of values for the different parameters is referred to as a *sample* or *sample point*. A collection of samples is referred to as a *design space*, *parameter sample space*, or, simply, *sample space*. You evaluate the optimization cost function for each point in the sample space. Then, you analyze the relation between the parameter value variations and the cost function value variations to understand how the parameters influence the cost function.

Each model evaluation has a computational expense and can be time intensive. Therefore, ideally, you want to use the smallest sample set that yields useful results. You can use techniques such as *design of experiments* (DOE) (also referred to as *experimental design*) to choose an efficient sample set for sensitivity analysis.

Use `sdo.ParameterSpace` to define the parameter space. This object specifies the probability distributions and correlations for the parameters. Use this object as an input to `sdo.sample` to generate samples from the specified parameter space.

Common considerations for parameter sampling include:

- "Probability Distribution" on page 4-4
- "Bounds" on page 4-5
- "Number of Samples" on page 4-5
- "Method of Sampling" on page 4-5
- "Custom Sample Sets" on page 4-7

Probability Distribution

Specify the probability distribution function that is best suited for each parameter. Use your knowledge of the system (empirical or theoretical) to choose the probability distribution for the parameter.

The Simulink Design Optimization software allows you to specify uniform (default) and normal distributions. If you have a Statistics and Machine Learning Toolbox license, you can also specify any univariate probability distribution that the toolbox provides.

Specify the probability distribution of a parameter using the `ParameterDistributions` property of an `sdo.ParameterSpace` object.

Bounds

Specify the upper and lower bounds of the value of a parameter. These bounds define the sampling range for the parameter. Use your knowledge of the parameter's range of likely values to choose the bounds.

To specify the bounds of a parameter, use the `Minimum` and `Maximum` properties of a `param.Continuous` object. Use this updated `param.Continuous` object when you specify the parameter space using an `sdo.ParameterSpace` object.

Number of Samples

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.

Specify the number of samples as the second input argument of `sdo.sample`.

Method of Sampling

After specifying the probability distributions and bounds for the parameters, you generate samples for the specified parameter space. You can specify the method used to generate these samples using the `Method` property of an `sdo.SampleOptions` object. Use this object as an input to `sdo.sample` to specify the sampling options. Specify the sampling method as one of the following:

- `'random'` — Random samples drawn from the probability distributions specified for the parameters.

  Suppose you specified a value for the `RankCorrelation` property of the `sdo.ParameterSpace` object that you use for sampling. The software uses the Iman-Conover method to impose the parameter correlations.

- `'lhs'` — Latin hypercube samples drawn from the probability distributions specified for the parameters. Use this option for a more systematic space-filling approach than random sampling. The following figure shows the difference between random sampling and Latin hybercube design-based sampling.
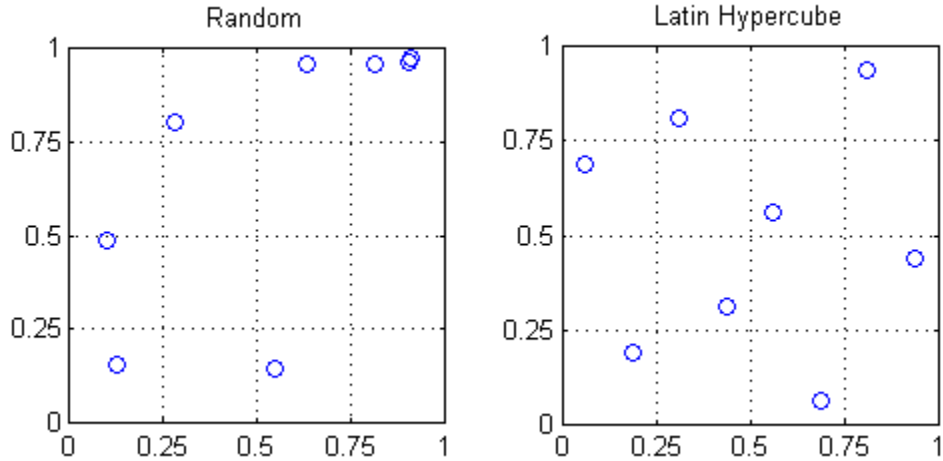
The figure shows 8 samples for 2 parameters, drawn from a uniform distribution, in the interval from 0 to 1. Random sampling can result in the clustering of samples (see the top right-hand corner of the plot). Latin hypercube designs, with their stratified approach to sampling, are better able to avoid such clustering.

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.

This method requires a Statistics and Machine Learning Toolbox license.

- `'copula'` — Random samples drawn from a copula. Use this option to impose correlations between the parameters.

You can use either a Gaussian copula (default) or a t copula. Specify the choice of copula using the `MethodOptions` property of the `sdo.SampleOptions` object. Use t copulas when the probability of extreme parameter values is not negligible. You must specify the degrees of freedom for a t copula. As you increase the degrees of freedom, the t copula converges to the Gaussian copula, and the probability of

extreme parameter values becomes negligible. The following figure shows 1000 samples drawn for 2 parameters, in the interval from 0 to 1, using the Gaussian and t copulas.
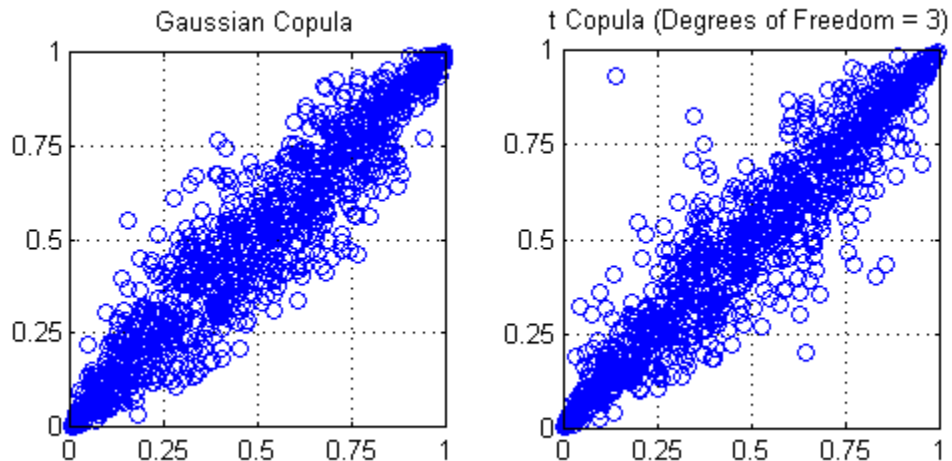
In comparison to the Gaussian copula, the t copula has more samples that represent the extreme values of the parameters.

You can specify the correlation type as either Spearman's rank correlation or Kendall's rank correlation.

For the `'copula'` method, you must also specify the value of the `RankCorrelation` property of the `sdo.ParameterSpace` object that you use for sampling.

This method requires a Statistics and Machine Learning Toolbox license.

Custom Sample Sets

You can specify a custom sample set. For example, suppose you want to generate a 1000 samples of the model parameter, R, which is a resistor.

```
R = param.Continuous('R',10);
```

Sample R in the 5% range of its nominal value. However, resistors of 1% tolerance are removed. So, you do not need to sample R in the 1% range. You can use the following approaches:

- "Specify Customized Probability Distribution" on page 4-8
- "Create Table of Custom Samples" on page 4-9

To visualize the samples and validate the sample space, use `sdo.scatterPlot`.

Specify Customized Probability Distribution

Use a customized probability distribution to configure the parameter space and generate samples.

You can use tools provided by the Statistics and Machine Learning Toolbox software to create customized probability distributions. For example,

```
x = [0.95 0.99 1.01 1.05]*R.Value;
F = [0 0.5 0.5 1];

pdR = makedist('PiecewiseLinear','x',x,'Fx',F);

x = linspace(.9*R.Value, 1.1*R.Value, 1e3);
plot(x, pdf(pdR, x));
```
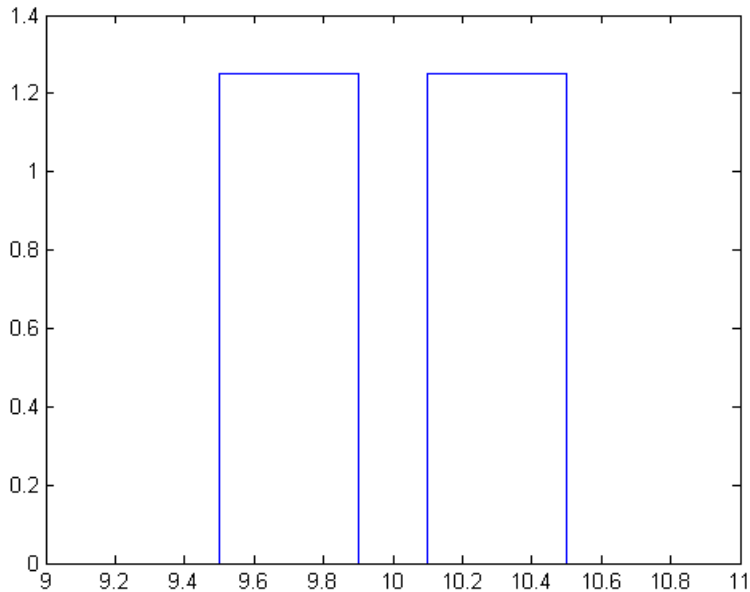
The call to `makedist` specifies a piecewise linear distribution for the resistor value, with a "hole" in the 1% range.

Specify `pdR` as the probability distribution for the R parameter when you create the `sdo.ParameterSpace` object to define the parameter space.

```
ps = sdo.ParameterSpace(R,pdR);
```

Generate the samples using `sdo.sample`.

```
x = sdo.sample(ps,1000);
```

Create Table of Custom Samples

Create a table of the custom samples. Specify one column for each parameter, and one row for each sample. The column name must be the same as the parameter name.

For example:

```
Rval = R.Value;
```

```
Ns = 1000;
x = table([linspace(.95*Rval,.99*Rval,Ns*.5) linspace(1.01*Rval,1.05*Rval,Ns*.5)]','Var
```

Consider another example, where you have two model parameters, A and B.

```
A = param.Continuous('A',1);
B = param.Continuous('B',10);
```

Vary A for the following values: 2,3,4. Vary B for the following values: 20,30,40. Generate a table of samples for every combination of A and B.

```
Avals = [2 3 4];
Bvals = [20 30 40];
[Agrid,Bgrid] = meshgrid(Avals,Bvals);
x = table(Agrid(:),Bgrid(:),'VariableNames',{'A','B'});
```

See Also
sdo.SampleOptions | sdo.sample

Related Examples
- "Design Exploration using Parameter Sampling (Code)" on page 4-17

More About
- "What Is Sensitivity Analysis?" on page 4-2

Sensitivity Analysis Methods

To analyze how the parameters and states (collectively referred to as *parameters*) of a Simulink model influence the cost function, first generate samples of the parameters. Then, evaluate the cost function for each sample. Finally, analyze the relationship between the parameter variations and the cost function values. Perform this analysis in the following ways:

- "Visual Analysis" on page 4-11
- "Quantitative Analysis" on page 4-11

Visual Analysis

Plot the cost function evaluations against the parameter samples to identify trends. This method is informal and provides visual intuition about how the various parameters affect the cost function.

You can use tools such as:

- `sdo.scatterPlot` — Scatter plot of the parameter samples against the cost function evaluation
- `surf`, `mesh` — 3-D plot of samples of two parameters against the cost function evaluation

Quantitative Analysis

Obtain summary statistics using `sdo.analyze`. This function performs linear correlation analysis by default. You can specify other analysis method(s) using an `sdo.AnalyzeOptions` options object.

Available analysis methods include:

Method Name	Description
Correlation	Use to analyze how a model parameter and the cost function are correlated
Partial correlation (requires a Statistics and Machine Learning Toolbox license)	Use to analyze how a model parameter and the cost function are correlated, removing the effects of the remaining parameters

Method Name	Description
Standardized regression	Use when you expect that the model parameters linearly influence the cost function

For information about the formulations of these methods, see `sdo.AnalyzeOptions`.

Linear vs. Ranked Analysis

Each of these methods supports the following analysis types:

- Linear analysis, also referred to as *Pearson* analysis — Uses raw data for analysis. Best suited when you expect a linear relation between the parameters and cost function.

- Ranked analysis, also referred to as *Spearman* analysis and *ranked transformation* — Uses ranks of data for analysis. Best suited when you expect a nonlinear monotonic relation between the parameters and the cost function.

For an example of ranked analysis, suppose you had the following data set:

x_1	x_2	y
9	20	340
5	60	106
2.3	50.4	870.5

Here x_1 and x_2 are model parameters, and y is the cost function. Each row represents a sample and the associated cost function evaluation.

The data is ranked on a per column basis. For example, when you rank the data in column 1 (x_1), which contains the entries 9, 5, and 2.3, the ranked data is equal to 3, 2, and 1. The ranked data set for the samples of x_1, x_2 and y are as follows:

x_1	x_2	y
3	1	2
2	3	1
1	2	3

The ranked data set can be used for correlation, partial correlation, or standardized regression analysis.

Linear analysis retains information about intervals between data values, whereas ranked analysis does not.

See Also
sdo.AnalyzeOptions | sdo.analyze | sdo.evaluate | sdo.sample

Related Examples
·   "Identify Key Parameters for Estimation (Code)" on page 4-33

More About
·   "What Is Sensitivity Analysis?" on page 4-2

How to Use Parallel Computing for Sensitivity Analysis

In this section...
"Configure Your System for Parallel Computing" on page 4-14
"Model Dependencies" on page 4-14
"Perform Sensitivity Analysis Using Parallel Computing (Code)" on page 4-15
"Troubleshooting" on page 4-16

Configure Your System for Parallel Computing

To perform global sensitivity analysis, you sample the model parameters and states (collectively referred to as *parameters*), and evaluate the cost function for each sample. You use `sdo.evaluate` to perform the cost function evaluation (also referred to as *model evaluation*). Evaluating the model for many samples can be time consuming. You can speed up the performance of `sdo.evaluate` using parallel computing on multicore processors or multiprocessor networks.

When you call `sdo.evaluate` with the parallel computing option enabled, the software uses the available parallel pool. If none is available, and **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, the software starts a parallel pool using the settings in those preferences. To open a parallel pool that uses a specific cluster profile, use:

```
parpool(MyProfile);
```

`MyProfile` is the name of a cluster profile.

For information regarding creating a cluster profile, see "Create and Modify Cluster Profiles" in the Parallel Computing Toolbox documentation.

Model Dependencies

Model dependencies are any referenced models, data such as model variables, S-functions, or additional files necessary to run the model. Before starting the parallel model evaluation, verify that the model dependencies are complete. Otherwise, you may get unexpected results.

Making File Dependencies Accessible to Remote Workers

When you use parallel computing, the Simulink Design Optimization software helps you identify model dependencies. To do so, the software uses the Simulink Manifest Tools. The dependency analysis may not find all the files required by your model. To learn more, see "Scope of Dependency Analysis" in the Simulink documentation. If your model has dependencies that are undetected or inaccessible by the parallel pool workers, then add them to the list of model dependencies.

The dependencies 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.
- Path dependencies: the paths to the model dependencies are specified to the parallel pool workers. If you are working in a multi-platform scenario, ensure that the paths are compatible across platforms.

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.

Perform Sensitivity Analysis Using Parallel Computing (Code)

To evaluate a model using parallel computing:

1 Ensure that the software can access parallel pool workers that use the appropriate cluster profile.

   For more information, see "Configure Your System for Parallel Computing" on page 4-14.

2 Open the model.

3 Specify the cost function and generate parameter samples for sensitivity analysis. For example, see "Design Exploration using Parameter Sampling (Code)".

4 Enable parallel computing using an evaluation option set.

   ```
   opt = sdo.EvaluateOptions;
   opt.UseParallel = 'always';
   ```

5 Find the model dependencies.

```
[dirs, files] = sdo.getModelDependencies(modelname)
```

Note: `sdo.getModelDependencies` may not detect all the dependencies in your model. For more information, see "Model Dependencies" on page 4-14. In this case, add the undetected dependencies manually.

---

6 Modify `files` to include any file dependencies that `sdo.getModelDependencies` does not detect.

```
files = vertcat(files,'C:\matlab\work\filename.m')
```

Note: If you do not want to copy the files to the remote workers, add any undetected path dependencies to `dirs` and update the paths on local drives to make them accessible to remote workers. See `sdo.getModelDependencies` for more details.

---

7 Add the file dependencies for evaluation.

```
opt.ParallelFileDependencies = files;
```

8 Specify the name of the model to be evaluated in parallel.

```
opt.EvaluatedModel = modelname;
```

9 Evaluate the model.

```
[pOpt,opt_info] = sdo.evaluate(fcn,samples,opt);
```

Troubleshooting

Why Don't I See the Evaluation Speed up I Expected Using Parallel Computing?

When you evaluate a model that does not have a large number of evaluations or does not take long to simulate, you might not see a speedup in the evaluation time. In such cases, the overhead associated with creating and distributing the parallel tasks outweighs the benefits of running the evaluation in parallel.

See Also

sdo.EvaluateOptions | parpool | sdo.evaluate | sdo.getModelDependencies

Design Exploration using Parameter Sampling (Code)

This example shows how to sample and explore a design space. You explore the design of a Continuously Stirred Tank Reactor to minimize product concentration variation and production cost. The design includes feed stock uncertainty.

You explore the CSTR design by characterizing design parameters using probability distributions. You use the distributions to generate random samples in the design space and perform Monte-Carlo evaluation of the design at these sample points. You then create plots to visualize the design space and select the best design. You can then use the best design as an initial guess for optimization of the design.

You can also use the sampled design space and Monte-Carlo evaluation output to analyze the influence of design parameters on the design, see "Sensitivity Analysis for Parameter Estimation (Code)"

Continuously Stirred Tank Reactor (CSTR) Model

Continuously Stirred Tank Reactors (CSTRs) are common in the process industry. The Simulink model, `sdoCSTR`, models a jacketed diabatic (i.e., non-adiabatic) tank reactor described in [1]. The CSTR is assumed to be perfectly mixed, with a single first-order exothermic and irreversible reaction, $A \rightarrow B$. A, the reactant, is converted to B, the product.

In this example, you use the following two-state CSTR model, which uses basic accounting and energy conservation principles:

$$\frac{dC_A}{dt} = \frac{F}{A*h}(C_{feed} - C_A) - r*C_A$$

$$\frac{dT}{dt} = \frac{F}{A*h}(T_{feed} - T) - \frac{H}{c_p\rho}r - \frac{U}{c_p*\rho*h}(T - T_{cool})$$

$$r = k_0 * e^{\frac{-E}{R*T}}$$

- C_A, and C_{feed} - Concentrations of A in the CSTR and in the feed [kgmol/m^3]

- T, T_{feed}, and T_{cool} - CSTR, feed, and coolant temperatures [K]

- F and ρ - Volumetric flow rate [m^3/h] and the density of the material in the CSTR [1/m^3]

- h and A - Height [m] and heated cross-sectional area [m^2] of the CSTR.

- k_0 - Pre-exponential non-thermal factor for reaction $A \to B$ [1/h]

- E and H - Activation energy and heat of reaction for $A \to B$ [kcal/kgmol]

- R - Boltzmann's gas constant [kcal/(kgmol * K)]

- c_p and U - Heat capacity [kcal/K] and heat transfer coefficients [kcal/(m^2 * K * h)]

Open the Simulink model.

```
open_system('sdoCSTR');
```


Copyright 2012 The MathWorks, Inc.

CSTR Design Problem

Assume that the CSTR is cylindrical, with the coolant applied to the base of the cylinder. Tune the CSTR cross-sectional area, A, and CSTR height, h, to meet the following design goals:

- Minimize the variation in residual concentration, C_A. Variations in the residual concentration negatively affect the quality of the CSTR product. Minimizing the variations also improves CSTR profit.

- Minimize the mean coolant temperature T_{cool}. Heating or cooling the jacket coolant temperature is expensive. Minimizing the mean coolant temperature improves CSTR profit.

The design must allow for variations in the quality of supply feed concentration, C_{feed}, and feed temperature, T_{feed}. The CSTR is fed with feed from different suppliers. The quality of the feed differs from supplier to supplier and also varies within each supply batch.

Specify Design Variables

Select the following model parameters as design variables:

- Cylinder cross-sectional area A

- Cylinder height h

```
p = sdo.getParameterFromModel('sdoCSTR',{'A','h'});
```

Limit the cross-sectional area to a range of [0.2 2] m^2.

```
p(1).Minimum = 0.2;
p(1).Maximum = 2;
```

Limit the height to a range of [0.5 3] m.

```
p(2).Minimum = 0.5;
p(2).Maximum = 3;
```

Sample the Design Space

Create a parameter space for the design variables. The parameter space characterizes the allowable parameter values and combinations of parameter values.

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

The parameter space uses default uniform distributions for the design variables. The distribution lower and upper bounds are set to the design variable minimum and maximum value respectively.

Use the `sdo.sample` function to generate samples from the parameter space. You use the samples to evaluate the model and explore the design space.

```
rng default; % For reproducibility
```

4-19

```
pSmpl = sdo.sample(pSpace,30);
```

Use the `sdo.scatterPlot` command to visualize the sampled parameter space. The scatter plot shows the parameter distributions on the diagonal subplots and pairwise parameter combinations on the off diagonal subplots.

```
figure, sdo.scatterPlot(pSmpl)
```


Specify Uncertain Variables

Select the feed concentration and feed temperature as uncertain variables. You evaluate the design using different values of feed temperature and concentration.

```
pUnc = sdo.getParameterFromModel('sdoCSTR',{'FeedCon0','FeedTemp0'});
```

Create a parameter space for the uncertain variables. Use normal distributions for both variables. Specify the mean as the current parameter value. Specify a variance of 5% of the mean for the feed concentration and 1% of the mean for the temperature.

```
uSpace = sdo.ParameterSpace(pUnc);
uSpace = setDistribution(uSpace,'FeedCon0',makedist('normal',pUnc(1).Value,0.05*pUnc(1)
uSpace = setDistribution(uSpace,'FeedTemp0',makedist('normal',pUnc(2).Value,0.01*pUnc(2
```

The feed concentration is inversely correlated with the feed temperature. Add this information to the parameter space.

```
uSpace.RankCorrelation = [1 -0.6; -0.6 1];
```

The rank correlation matrix has a row and column for each parameter with the (i,j) entry specifying the correlation between the i and j parameters.

Sample the parameter space. The scatter plot shows the correlation between concentration and temperature.

```
uSmpl = sdo.sample(uSpace,60);
sdo.scatterPlot(uSmpl)
```
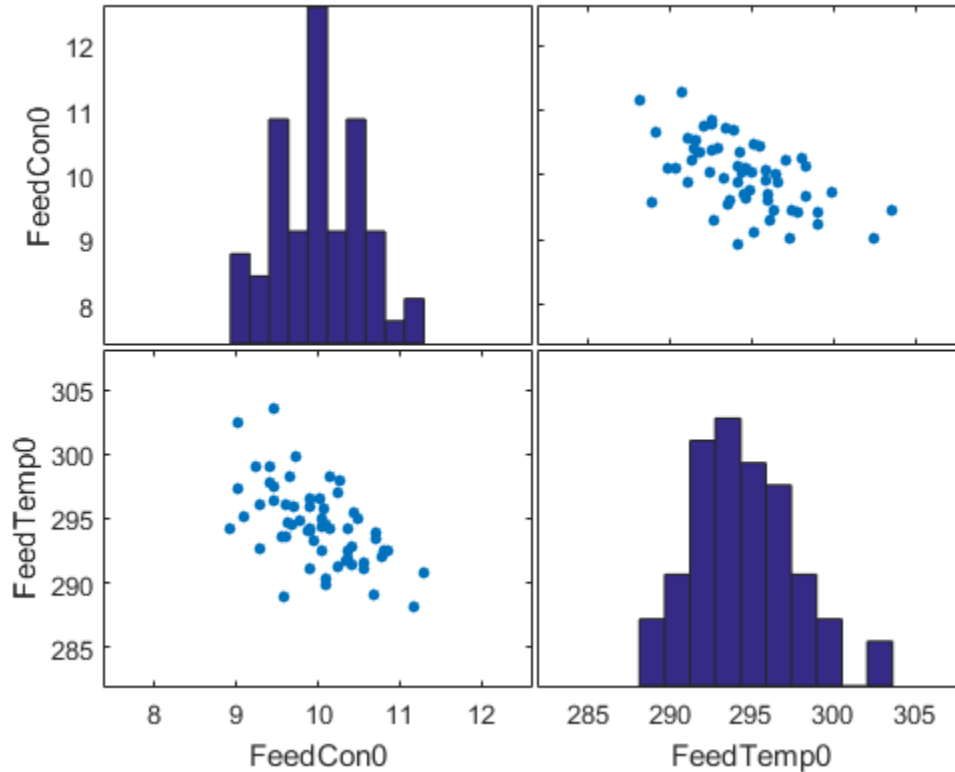
Ideally you want to evaluate the design for every combination of points in the design and uncertain spaces, which implies 30*60 = 1800 simulations. Each simulation takes around 0.5 sec. You can use parallel computing to speed up the evaluation. For this example you instead only use the samples that have maximum & minimum concentration and temperature values, reducing the evaluation time to around 1 min.

```
[~,iminC] = min(uSmpl.FeedCon0);
[~,imaxC] = max(uSmpl.FeedCon0);
[~,iminT] = min(uSmpl.FeedTemp0);
[~,imaxT] = max(uSmpl.FeedTemp0);
uSmpl = uSmpl(unique([iminC,imaxC,iminT,imaxT]) ,:)
```

```
uSmpl =
```

```
FeedCon0    FeedTemp0
_____    _____

9.4555      303.58
11.175      288.13
11.293      290.73
8.9308      294.16
```

Create Evaluation Function

Create a function that evaluates the design for a given sample point in the design space. The design is evaluated on how well it minimizes the variation in residual concentration and mean coolant temperature.

Specify Design Requirements

Evaluating a point in the design space requires logging model signals. Logged signals are used to evaluate the design requirements.

Log the following signals:

• CSTR concentration, available at the second output port of the sdoCSTR/CSTR block

```
Conc = Simulink.SimulationData.SignalLoggingInfo;
Conc.BlockPath              = 'sdoCSTR/CSTR';
Conc.OutputPortIndex        = 2;
Conc.LoggingInfo.NameMode   = 1;
Conc.LoggingInfo.LoggingName = 'Concentration';
```

• Coolant temperature, available at the first output of the sdoCSTR/Controller block

```
Coolant = Simulink.SimulationData.SignalLoggingInfo;
Coolant.BlockPath              = 'sdoCSTR/Controller';
Coolant.OutputPortIndex        = 1;
Coolant.LoggingInfo.NameMode   = 1;
Coolant.LoggingInfo.LoggingName = 'Coolant';
```

Create and configure a simulation test object to log the required signals.

```
simulator = sdo.SimulationTest('sdoCSTR');
simulator.LoggingInfo.Signals = [Conc,Coolant];
```

Evaluation Function

Use an anonymous function with one argument that calls the `sdoCSTR_design` function.

```
evalDesign = @(p) sdoCSTR_design(p,simulator,pUnc,uSmpl);
```

The `evalDesign` function:

- Has one input argument that specifies the CSTR dimensions

- Returns the optimization objective value

The `sdoCSTR_design` function uses a `for` loop that iterates through the sample values specified for the feed concentration and temperature. Within the loop, the function:

- Simulates the model using the current design point, feed concentration, and feed temperature values

- Calculates the residual concentration variation and coolant temperature costs

To view the objective function, type `edit sdoCSTR_design`.

type sdoCSTR_design

```
function design = sdoCSTR_design(p,simulator,pUnc,smplUnc)
%SDOCSTR_DESIGN
%
% The sdoCSTR_design function is used to evaluate a CSTR  design.
%
% The |p| input argument is the vector of CSTR dimensions.
%
% The |simulator| input argument is a sdo.SimulinkTest object used to
% simulate the |sdoCSTR| model and log simulation signals.
%
% The |pUnc| input argument is a vector of parameters to specify the CSTR
% input feed concentration and feed temperature. The |smplUnc| argument is
% a table of different feed concentration and temperature values.
%
% The |design| return argument contains information about the design
% evaluation that can be used by the |sdo.optimize| function to optimize
% the design.
%
% see also sdo.optimize, sdoExampleCostFunction
%
% Copyright 2012-2013 The MathWorks, Inc.
```

```
%% Model Simulations and Evaluations
%
% For each value in |smplUnc|, configure and simulate the model. Use
% the logged concentration and coolant signals to compute the design cost.
%
costConc    = 0;
costCoolant = 0;
for ct=1:size(smplUnc,1)
    %Set the feed concentration and temperature values
    pUnc(1).Value = smplUnc{ct,1};
    pUnc(2).Value = smplUnc{ct,2};

    %Simulate model
    simulator.Parameters = [p; pUnc];
    simulator = sim(simulator);
    logName   = get_param('sdoCSTR','SignalLoggingName');
    simLog    = get(simulator.LoggedData,logName);

    %Compute Concentration cost based on the standard deviation of the
    %concentration from a nominal value.
    Sig = find(simLog,'Concentration');
    costConc = costConc+10*std(Sig.Values-2);

    %Compute coolant cost based on the mean deviation from room
    %temperature.
    Sig = find(simLog,'Coolant');
    costCoolant = costCoolant+abs(mean(Sig.Values - 294))/30;
end

%% Return Total Cost
%
% Compute the total cost as a sum of the concentration and coolant costs.
%
design.F = costConc + costCoolant;

%%
% Add the individual cost terms to the return argument. These are not used
% by the optimizer, but included for convenience.
design.costConc    = costConc;
design.costCoolant = costCoolant;
end
```

Evaluate

Use the `sdo.evaluate` command to evaluate the model at the sample design points.

```
y = sdo.evaluate(evalDesign,p,pSmpl);
```

Model evaluated at 30 samples.

View the results of the evaluation using a scatter plot. The scatter plot shows pairwise plots for each design variable (A,h) and design cost. The plot include the total cost, F, as well as coolant and concentration costs, `costCoolant` and `costConc` respectively.

```
sdo.scatterPlot(pSmpl,y);
```
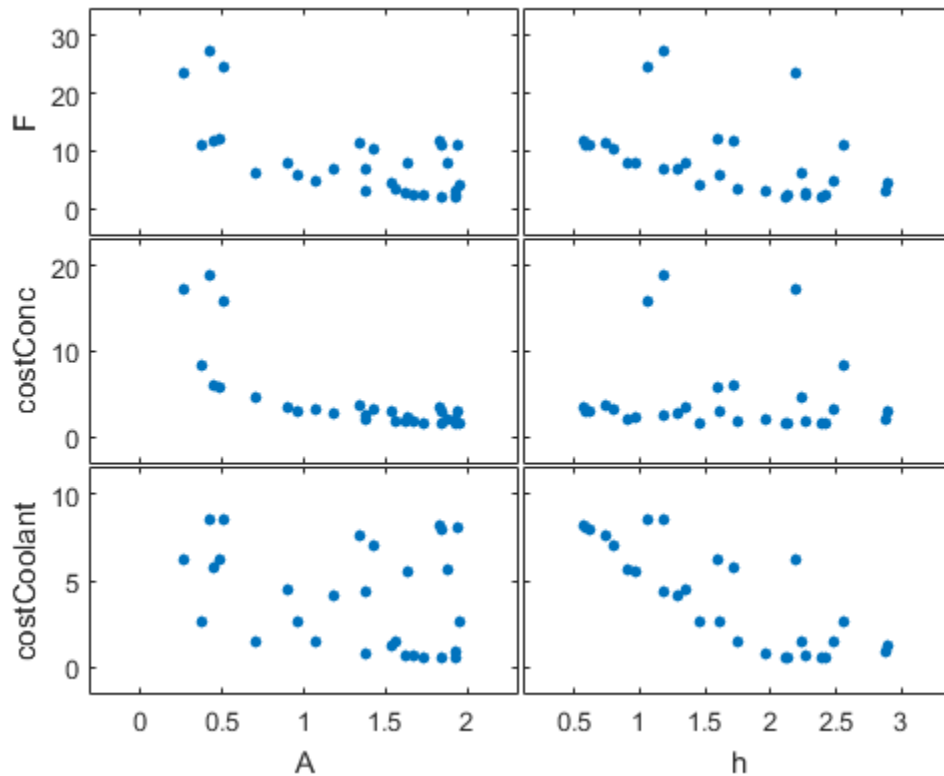
The plot shows that larger cross-sectional areas result in lower total costs. However it is difficult to tell how the height influences the total cost.

Create a mesh plot showing the total cost as a function of A and h.

```
Ftotal = scatteredInterpolant(pSmpl.A,pSmpl.h,y.F);
xR = linspace(min(pSmpl.A),max(pSmpl.A),60);
yR = linspace(min(pSmpl.h),max(pSmpl.h),60);
[xx,yy] = meshgrid(xR,yR);
zz = Ftotal(xx,yy);
mesh(xx,yy,zz)
view(56,30)
title('Total cost as function of A and h')
zlabel('Ftotal')
xlabel(p(1).Name), ylabel(p(2).Name);
```

Total cost as function of A and h
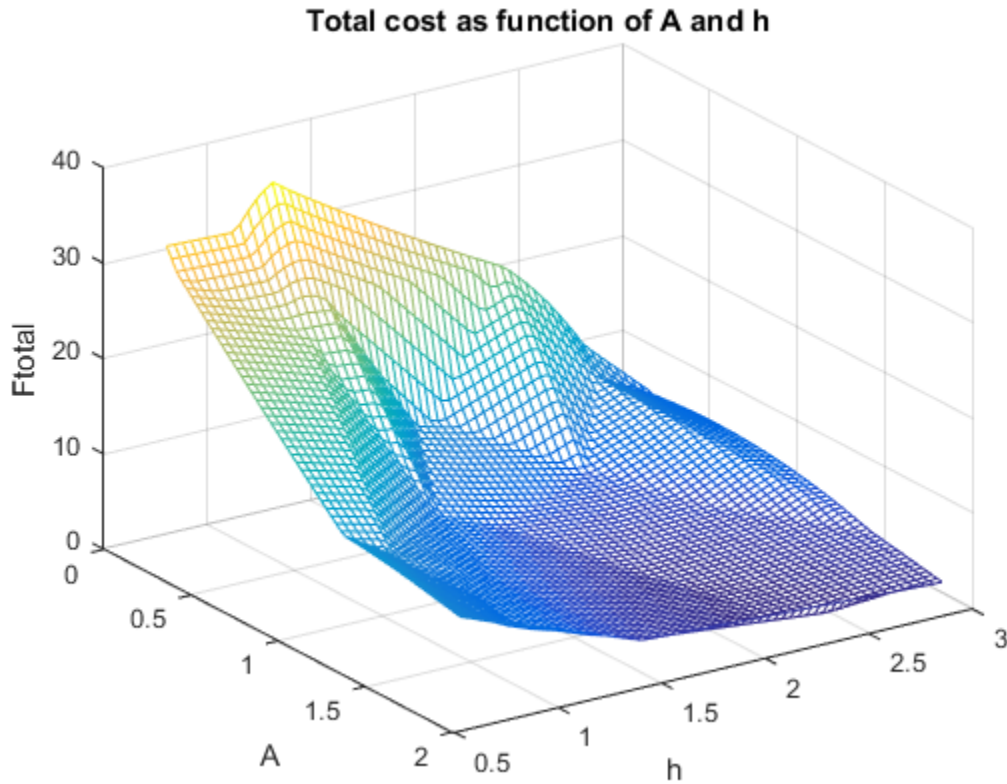
The plot shows that high values of A and h result in lower costs. The best design in the sampled space corresponds to the design with the lowest cost value.

```
[~,idx] = min(y.F);
pBest = [y(idx,:), pSmpl(idx,:)]
```

```
pBest =
```

F	costConc	costCoolant	A	h
2.1052	1.5757	0.52953	1.8483	2.1158

Refine the Design Space

The total cost surface plot shows that low cost designs are designs with A in the range [1.5 2] and h in the range [2 3]. Modify the parameter space distributions for A and h and resample the design space to focus on this region.

```
pSpace = setDistribution(pSpace,'A',makedist('uniform',1.5,2));
pSpace = setDistribution(pSpace,'h',makedist('uniform',2,3));
pSmpl = sdo.sample(pSpace,30);
```

Add the `pBest` found earlier to the new samples so that it is part of the refined design space.

```
pSmpl = [pSmpl;pBest(:,4:5)];
sdo.scatterPlot(pSmpl)
```
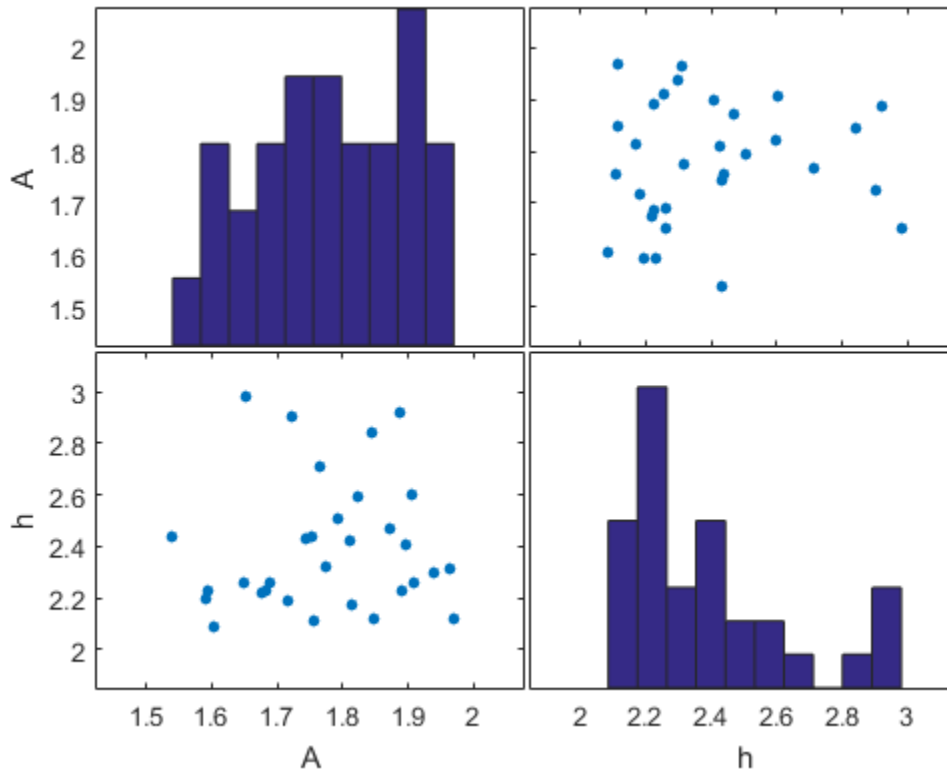
Evaluate using Refined Design Space

```
y = sdo.evaluate(evalDesign,p,pSmpl);
```

Model evaluated at 31 samples.

Create a mesh plot for this section of the design space. The surface indicates that better designs are near the A = 1.9, h = 2.1 point.

```
Ftotal = scatteredInterpolant(pSmpl.A,pSmpl.h,y.F);
xR = linspace(min(pSmpl.A),max(pSmpl.A),60);
yR = linspace(min(pSmpl.h),max(pSmpl.h),60);
[xx,yy] = meshgrid(xR,yR);
zz = Ftotal(xx,yy);
mesh(xx,yy,zz)
view(56,30)
title('Total cost as function of A and h')
zlabel('Ftotal')
xlabel(p(1).Name), ylabel(p(2).Name);
```

Total cost as function of A and h
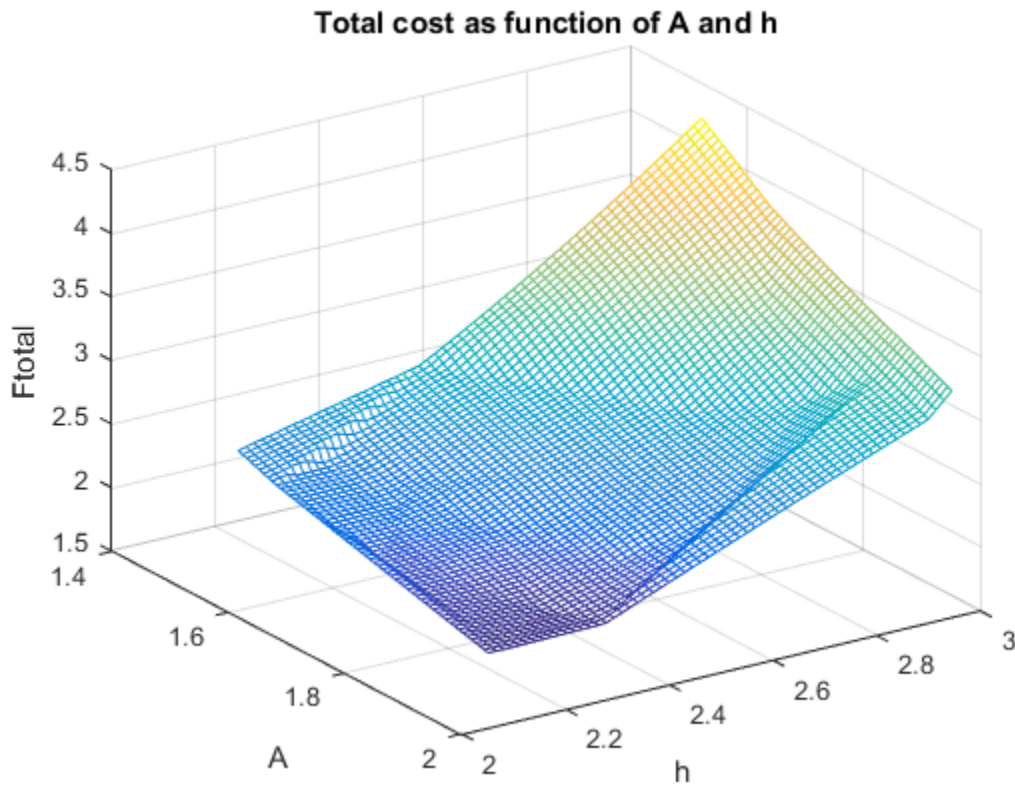
Find the best design from the new design space and compare with the best design point found earlier.

```
[~,idx] = min(y.F);
pBest = [pBest; [y(idx,:), pSmpl(idx,:)]]
```

```
pBest =

       F       costConc    costCoolant      A        h
     _____    _____    _____    _____    _____

     2.1052     1.5757        0.52953      1.8483    2.1158
      1.979     1.4838        0.49528      1.9695    2.1174
```

4-31

The best design in the refined design space is better than the design found earlier. This indicates that there may be better designs in the same region and warrants refining the design space further. Alternatively you can use the best design point as an initial guess for optimization.

Related Examples

To learn how to optimize the CSTR design using the `sdo.optimize` command, see "Design Optimization with Uncertain Variables (Code)".

To learn how to analyze the influence of design parameters on the design using the `sdo.analyze` command, see "Sensitivity Analysis for Parameter Estimation (Code)"

References

[1] Bequette, B.W. *Process Dynamics: Modeling, Analysis and Simulation.* 1st ed. Upper Saddle River, NJ: Prentice Hall, 1998.

Close the model

```
bdclose('sdoCSTR')
```

Identify Key Parameters for Estimation (Code)

This example shows how to use sensitivity analysis to narrow down the number of parameters that you need to estimate to fit a model. This example uses a model of the vestibulo-ocular reflex, which generates compensatory eye movements.

Model Description

The vestibulo-ocular reflex (VOR) enables the eyes to move at the same speed and in the opposite direction as the head, so that vision is not blurred when the head moves during normal activity. For example, if the head turns in one direction, the eyes turn in the opposite direction, with the same speed. This happens even in the dark. In fact, the VOR is most easily characterized by measurements in the dark, to ensure that eye movements are predominantly driven by the VOR.

The file sdoVOR_Data.mat contains uniformly sampled data of stimulation and eye movements. If the VOR were perfectly compensatory, then a plot of eye movement data, when flipped vertically, would overlay exactly on top of a plot of head motion data. Such a system would be described by a gain of 1 and a phase of 180 degrees. However, when we plot the data in the file sdoVOR_Data.mat, the eye movements are close, but not perfectly compensatory.

```
load sdoVOR_Data.mat;   % Column vectors:   Time  HeadData  EyeData
figure
plot(Time, HeadData, ':b',    Time, EyeData, '-g')
xlabel('Time (sec)')
ylabel('Angular  Velocity  (deg/sec)')
ylim([-110  110])
legend('Head Data', 'Eye Data')
```
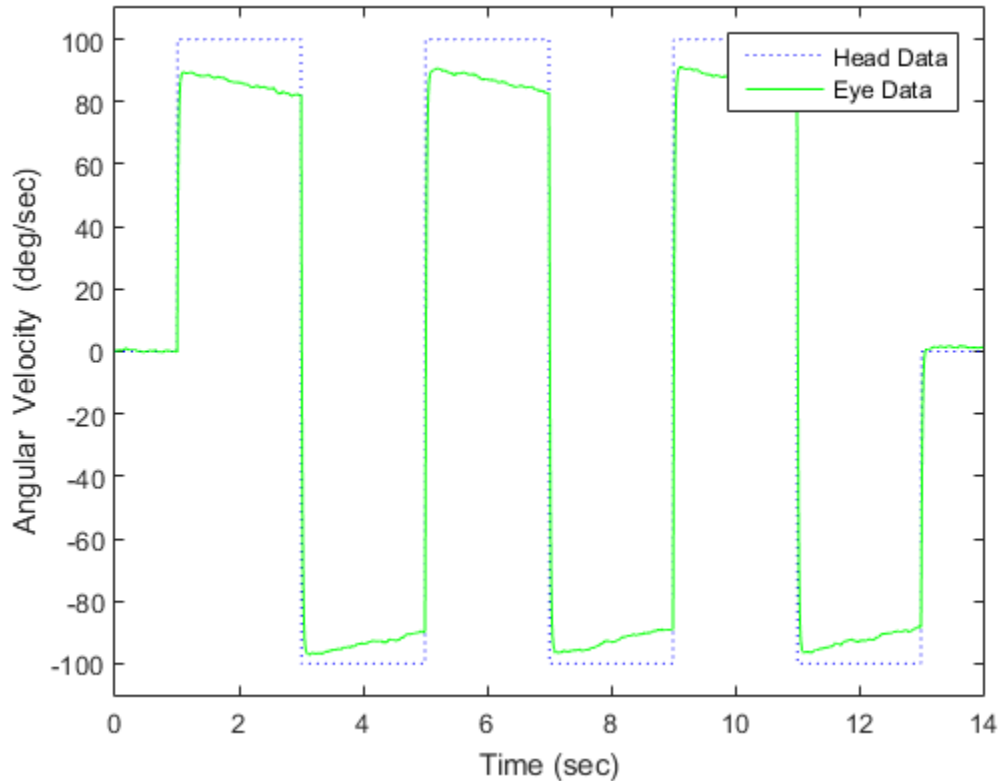
The eye movement data does not perfectly overlay the head motion data, and this can be modeled by several factors. Head rotation is sensed by organs in the inner ears, known as semicircular canals. These detect head motion and transmit signals about head motion to the brain, which sends motor commands to the eye muscles, so that eye movements compensate for head motion. We would like to use this eye movement data to estimate the parameters in the models for these various stages. The model we will use is shown below. There are four parameters in the model: `Delay`, `Gain`, `Tc`, and `Tp`.

```
model_name = 'sdoVOR';
open_system(model_name)
```

Vestibulo-Ocular Reflex
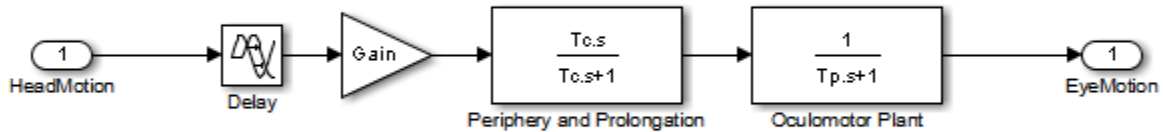
Copyright 2013 The MathWorks, Inc.

The Delay parameter models the fact that there is some delay in communicating the signals from the inner ear to the brain and the eyes. This delay is due to the time needed for chemical neurotransmitters to traverse the synaptic clefts between nerve cells. Based on the number of synapses involved in the vestibulo-ocular reflex, this delay is expected to be around 5 ms. For estimation purposes, we will assume it is between 2 and 9 ms.

```
Delay = sdo.getParameterFromModel(model_name, 'Delay');
Delay.Value = 0.005;    % seconds
Delay.Minimum = 0.002;
Delay.Maximum = 0.009;
```

The Gain parameter models the fact that the the eyes do not move quite as much as the head does. We will use 0.8 as our initial guess, and assume it is between 0.6 and 1.

```
Gain = sdo.getParameterFromModel(model_name, 'Gain');
Gain.Value = 0.8;
Gain.Minimum = 0.6;
Gain.Maximum = 1;
```

The Tc parameter models the dynamics associated with the semicircular canals, as well as some additional neural processing. The canals are high-pass filters, because after a subject has been put into rotational motion, the neurally active membranes in the canals slowly relax back to resting position, so the canals stop sensing motion. Thus in the plot above, after the stimulation undergoes transition edges, the eye movements tend to depart from the stimulation over time. Based on mechanical characteristics of the canals, combined with additional neural processing which prolongs this time constant to improve

the accuracy of the VOR, we will estimate the Tc parameter to be 15 seconds, and assume it is between 10 and 30 seconds.

```
Tc = sdo.getParameterFromModel(model_name, 'Tc');
Tc.Value = 15;
Tc.Minimum = 10;
Tc.Maximum = 30;
```

Finally, the Tp parameter models the dynamics of the oculomotor plant, i.e. the eye and the muscles and tissues attached to it. The plant can be modeled by two poles, however it is believed that the pole with the larger time constant is cancelled by precompensation in the brain, to enable the eye to make quick movements. Thus in the plot, when the stimulation undergoes transition edges, the eye movements follow with only a little delay. For the Tp parameter, we will use 0.01 seconds as our initial guess, and assume it is between 0.005 and 0.05 seconds.

```
Tp = sdo.getParameterFromModel(model_name, 'Tp');
Tp.Value = 0.01;
Tp.Minimum = 0.005;
Tp.Maximum = 0.05;
```

Collect these parameters into a vector.

```
v = [Delay Gain Tc Tp];
```

Compare Measured Data to Initial Simulated Output

Create an Experiment object. Specify `HeadData` as input.

```
Exp = sdo.Experiment(model_name);
Exp.InputData = timeseries(HeadData, Time);
```

Associate eye movement data with model output.

```
EyeMotion = Simulink.SimulationData.Signal;
EyeMotion.Name = 'EyeMotion';
EyeMotion.BlockPath = [model_name  '/Oculomotor Plant'];
EyeMotion.PortType = 'outport';
EyeMotion.PortIndex = 1;
EyeMotion.Values = timeseries(EyeData, Time);
```

Add `EyeMotion` to the experiment.

```
Exp.OutputData = EyeMotion;
```

Use the data's timing characteristics in the model.

```
stop_time = Time(end);
set_param(gcs, 'StopTime', num2str(stop_time));
dt = Time(2) - Time(1);
set_param(gcs, 'FixedStep', num2str(dt))
```

Create a simulation scenario using the experiment, and obtain the simulated output.

```
Exp = setEstimatedValues(Exp, v);    % use vector of parameters/states
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);
```

Search for the model_residual signal in the logged simulation data.

```
SimLog = find(Simulator.LoggedData, ...
 get_param(model_name, 'SignalLoggingName') );
EyeSignal = find(SimLog, 'EyeMotion');
```

The model output does not match the data very well, as shown by the residual, which we can compute by calling the objective function.

```
estFcn = @(v) sdoVOR_Objective(v, Exp, 'Residuals');
Model_Error = estFcn(v);
plot(Time, EyeData, '-g', ...
 EyeSignal.Values.Time, EyeSignal.Values.Data, '--c', ...
 Time, Model_Error.F, '-r');
xlabel('Time (sec)');
ylabel('Angular  Velocity  (deg/sec)');
legend('Eye Data', 'Model', 'Residual');
```
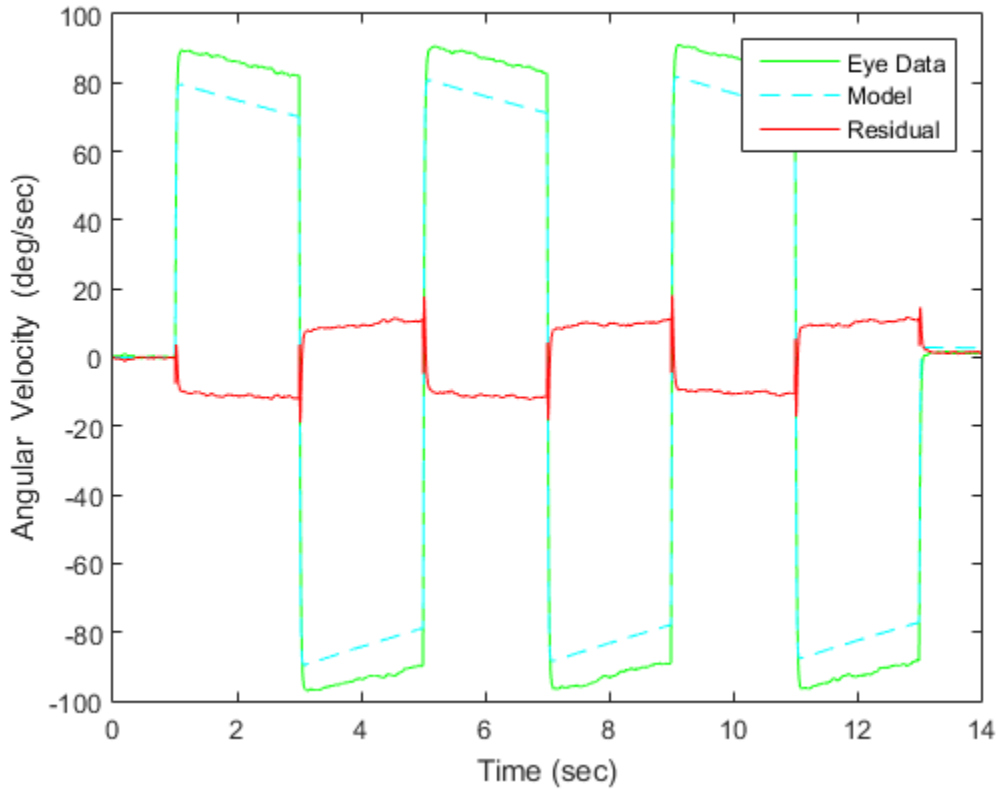
The objective function used above is defined in the file "sdoVOR_Objective.m".

```
type sdoVOR_Objective.m
```

```
function vals = sdoVOR_Objective(v, Exp, Method)
% Compare model output with data
%
%    Inputs:
%        v - vector of parameters and/or states
%        Exp - Experiment object
%        Method - 'SSE' for scalar output, 'Residuals' for vector of residuals

% Requirement setup
```

```
req = sdo.requirements.SignalTracking;
req.Type = '==';
req.Method = Method;

% If Residuals requested, keep on same scale as signals, for plotting
switch Method
 case 'Residuals'
  req.Normalize = 'off';
end

% Simulate the model
Exp = setEstimatedValues(Exp, v);    % use vector of parameters/states
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);

% Compare model output with data
SimLog = find(Simulator.LoggedData,  ...
 get_param(Exp.ModelName, 'SignalLoggingName') );
OutputModel = find(SimLog, 'EyeMotion');
Model_Error = evalRequirement(req, OutputModel.Values, Exp.OutputData.Values);
vals.F = Model_Error;
```
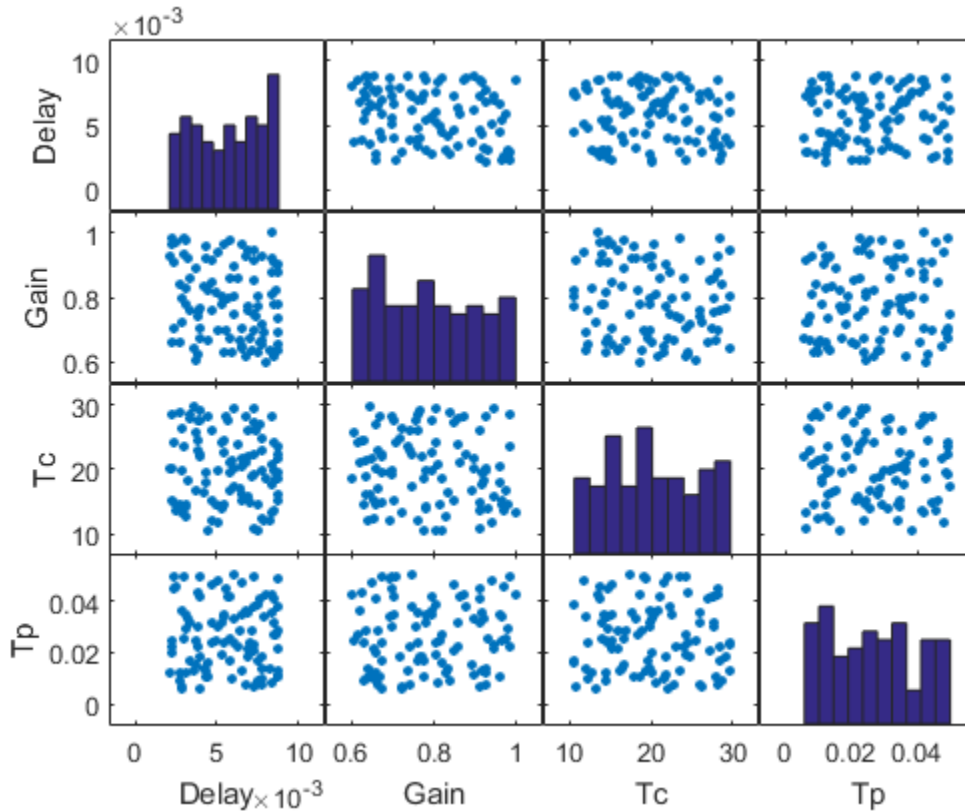
Sensitivity Analysis

Create an object to sample the parameter space.

```
ps = sdo.ParameterSpace([Delay ; Gain ; Tc ; Tp]);
```

Generate 100 samples from the parameter space.

```
rng default;    % for reproducibility
x  = sdo.sample(ps, 100);
sdo.scatterPlot(x);
```

The sampling above used default options, and these are reflected in the plots above. Parameter values were selected at random from distributions that were uniform over the range of each parameter. Consequently, the histogram plots along the diagonal appear approximately uniform. If Statistics and Machine Learning Toolbox™ is available, a number of distributions may be used in addition to uniform and normal, and sampling can be done in a Latin hypercube pattern.

The off-diagonal plots above show scatter plots between pairs of different variables. Since we did not specify a RankCorrelation matrix in ps, the scatter plots do not indicate correlations. However, if parameters were believed to be correlated, this can be specified using the RankCorrelation property of ps.

For sensitivity analysis, it is simpler to use a scalar objective, so we will specify the sum of squared errors, "SSE":

```
estFcn = @(v) sdoVOR_Objective(v, Exp, 'SSE');
y = sdo.evaluate(estFcn, ps, x);
```

Model evaluated at 100 samples.

Evaluation could also be sped up using parallel computing.

Obtain the standardized regression coefficients.

```
opts = sdo.AnalyzeOptions;
opts.Method = 'StandardizedRegression';
sensitivities = sdo.analyze(x, y, opts);
```

Other types of analysis include correlation and, if Statistics and Machine Learning Toolbox is available, partial correlation.

We can view the analysis results.

```
disp(sensitivities)

                    F
              _____

    Delay       0.01303
    Gain       -0.90873
    Tc         -0.044395
    Tp          0.19919
```

For standardized regression, parameters that highly influence the model output have sensitivity magnitudes close to 1. On the other hand, less influential parameters have smaller sensitivity magnitudes. We see that this objective function is sensitive to changes in the Gain and Tp parameters, but much less sensitive to changes in the Delay and Tc parameters.

You can validate sensitivity analysis results by resampling and reevaluating the objective function for the samples. You can also use engineering intuition for a quick analysis. For example, in this model, the time constant Tc ranges from 10 to 30 seconds. Even the minimum value of 10 seconds is large compared to the 2-second duration for which the head motion stimulation is held at constant velocity. Therefore, Tc is not expected to affect the output greatly. However, even when this kind of intuition

is not readily available in other models, sensitivity analysis can help highlight which parameters are influential.

Based on the results of sensitivity analysis, designate the `Delay` and `Tc` parameters as fixed when optimizing. This reduction in the number of free parameters speeds up optimization.

```
Delay.Free = false;
Tc.Free = false;
```

Optimization

We can use the minimum from sensitivity analysis as the initial guess for optimization.

```
[fval, idx_min] = min(y.F);
Delay.Value = x.Delay(idx_min);
Gain.Value = x.Gain(idx_min);
Tc.Value = x.Tc(idx_min);
Tp.Value = x.Tp(idx_min);
%
v = [Delay Gain Tc Tp];
opts = sdo.OptimizeOptions;
opts.Method = 'fmincon';
```

As was the case with model evaluations in sensitivity analysis, parallel computing could be used to speed up the optimization.

```
vOpt = sdo.optimize(estFcn, v, opts);
disp(vOpt)
```

```
 Optimization started 20-Jan-2015 08:45:40

                              max      Step-size    First-order
 Iter F-count        f(x)  constraint               optimality
    0      5      13.4798         0
    1     18      12.2052         0        0.129           305
    2     30      11.1441         0       0.0648           781
    3     41      10.0493         0       0.0843           289
    4     46      9.23607         0       0.0758           227
    5     51      8.76122         0       0.0183          10.1
    6     56      8.75862         0      0.00184         0.476
    7     57      8.75862         0     8.41e-05         0.476
Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
```

```
the selected value of the step size tolerance and constraints are
satisfied to within the selected value of the constraint tolerance.

(1,1) =

       Name: 'Delay'
      Value: 0.0038
    Minimum: 0.0020
    Maximum: 0.0090
       Free: 0
      Scale: 0.0078
       Info: [1x1 struct]


(1,2) =

       Name: 'Gain'
      Value: 0.9012
    Minimum: 0.6000
    Maximum: 1
       Free: 1
      Scale: 1
       Info: [1x1 struct]


(1,3) =

       Name: 'Tc'
      Value: 16.6833
    Minimum: 10
    Maximum: 30
       Free: 0
      Scale: 16
       Info: [1x1 struct]


(1,4) =

       Name: 'Tp'
      Value: 0.0157
    Minimum: 0.0050
    Maximum: 0.0500
       Free: 1
      Scale: 0.0156
```

```
        Info: [1x1 struct]


1x4 param.Continuous
```

Visualizing Result of Optimization

Obtain the model response after estimation. Search for the model_residual signal in the logged simulation data.

```
Exp = setEstimatedValues(Exp, vOpt);
Simulator = createSimulator(Exp);
Simulator = sim(Simulator);
SimLog = find(Simulator.LoggedData,  ...
 get_param(model_name, 'SignalLoggingName') );
EyeSignal = find(SimLog, 'EyeMotion');
```

Comparing the measured eye data with the optimized model response shows that the residuals are much smaller.

```
estFcn = @(v) sdoVOR_Objective(v, Exp, 'Residuals');
Model_Error = estFcn(vOpt);
plot(Time, EyeData, '-g',  ...
 EyeSignal.Values.Time, EyeSignal.Values.Data, '--c',  ...
 Time, Model_Error.F, '-r');
xlabel('Time (sec)');
ylabel('Angular  Velocity  (deg/sec)');
legend('Eye Data', 'Model', 'Residual');
```
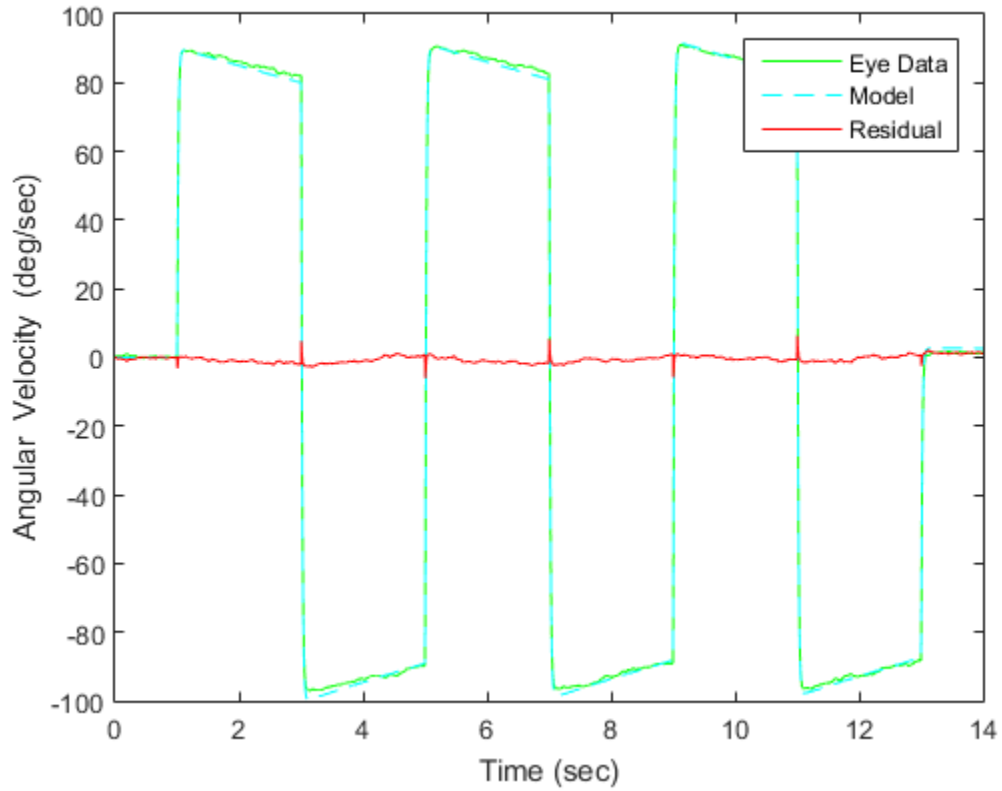
Close the model

```
bdclose(model_name)
```

5

Optimization-Based Control Design

Overview of Optimization-Based Compensator Design

You can design optimization-based controllers for Simulink models to meet time and frequency-domain design requirements, as described in "Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)" on page 3-94.

If you have Control System Toolbox software installed, you can also design and optimize control systems by tuning controller elements or parameters within a SISO Design Task in the Control and Estimation Tools Manager. You can tune elements or parameters such as poles, zeros, and gains within any controller in the system and optimize the open and closed loop responses to meet time- and frequency-domain requirements.

Optimize the responses of systems in the SISO Design Task to meet both time- and frequency-domain performance requirements by graphically constraining signals:

- Add frequency-domain design requirements to plots such as root-locus, Nichols, and Bode in the SISO Design Task graphical tuning editor called SISO Design Tool.
- Add time-domain design requirements to plots such as step or impulse response (when displayed within the Linear System Analyzer as part of a SISO Design Task).

You can use optimization methods in a SISO Design Task in the Control and Estimation Tools Manager to tune both command-line LTI models as well as Simulink models:

- Create an LTI model using the Control System Toolbox command-line functions and use the `controlSystemDesigner` function to create a SISO Design Task for the model. For an example, see "Optimize LTI System to Meet Frequency-Domain Requirements" on page 5-44.
- Use a Simulink Compensator Design task (from Simulink Control Design software) to automatically analyze the model and then create a SISO Design Task for a linearized version of the model. You can then use the optimization techniques in the SISO Design Task to tune the response of the linearized Simulink model. For an example, see "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)".

Note: When using response optimization within a SISO Design Task you cannot add uncertainty to system parameters.

When using a SISO Design Task, Simulink Design Optimization software automatically sets the model's simulation start and stop time and you cannot directly change them.

By default, the simulation starts at 0 and continues until the SISO Design Task determines that the dynamics of the model have settled out. In addition, when the design requirements extend beyond this point, the simulation continues to the extent of the design requirements. Although you cannot directly adjust the start or stop time of the simulation, you can adjust the design requirements to extend further in time and thus force the simulation to continue to a certain point.

Time-Domain Design Requirements in Simulink

Specify Piecewise-Linear Lower and Upper Bounds

To specify upper and lower bounds on a signal:

1   In the Response Optimization tool, select **Signal Bound** in the **New** drop-down list. A window opens where you specify upper or lower bounds on a signal.

2   Specify a requirement name in the **Name** box.

3   Select the requirement type using the **Type** list.

4   Specify the edge start and end times and corresponding amplitude in the **Time (s)** and **Amplitude** columns.

5   Click ⊞ to specify additional bound edges.

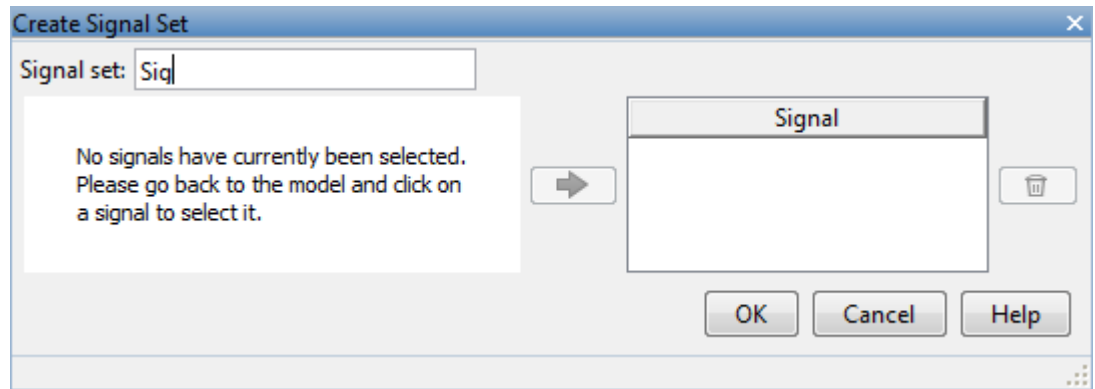   Select a row and click 🗑 to delete a bound edge.

6   In the **Select Signals to Bound** area, select a logged signal to apply the requirement to.

    If you have already selected signals, as described in "Specify Signals to Log" on page 3-12, they appear in the list. Select the corresponding check-box.

    If you haven't selected a signal to log:

    a   Click ⊞. A window opens where you specify the logged signal.

    b   In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click ⬜ to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

7 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can add a Check Custom Bounds block to your model to specify piecewise-linear bounds.
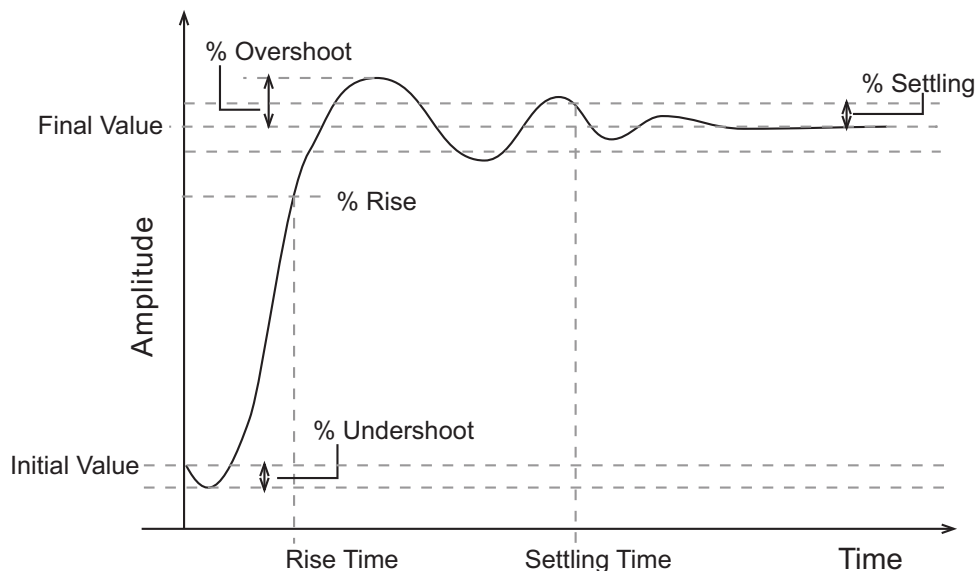
Specify Step Response Characteristics

To specify step response characteristics:

1 You can apply this requirement to either a signal or a linearization of your model.

In the Response Optimization Tool, click **New**. To apply this requirement to a signal, select the **Step Response Envelope** entry in the **New Time Domain Requirement** section of the **New** list. To apply this requirement to a linearization of your model, select the **Step Response Envelope** entry in the **New Frequency Domain Requirement** section of the **New** list. The latter option requires Simulink Control Design software.

A window opens where you specify the step response requirements on a signal, or system.

2 Specify a requirement name in the **Name** box.

3 Specify the step response characteristics:

- **Initial value**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final value**: Input level after the step occurs

- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.

- **% Rise**: The percentage used in the rise time.

- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.

- **% Settling**: The percentage used in the settling time.

- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

4 Specify the signals or systems to be bound.

You can apply this requirement to a model signal or to a linearization of your Simulink model (requires Simulink Control Design software).
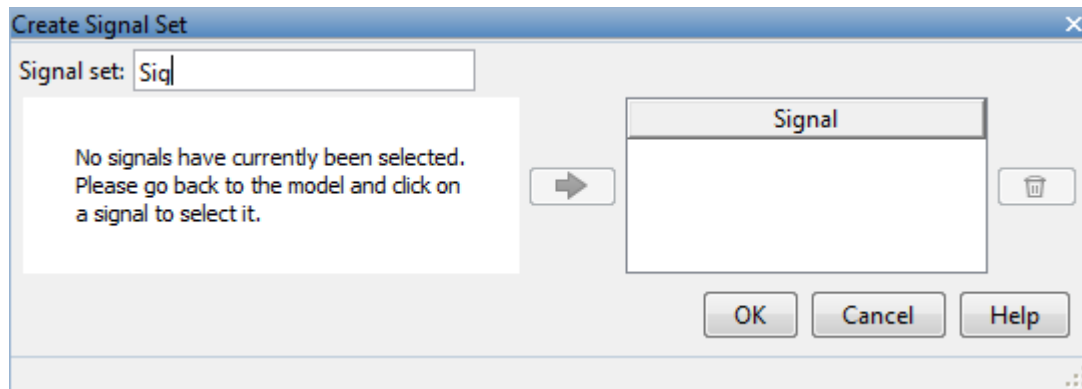
- Apply this requirement to a model signal:

  In the **Select Signals to Bound** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check-box.

  If you haven't selected a signal to log:

  a Click [+]. A window opens where you specify the logged signal.

  b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

- Apply this requirement to a linear system.

  Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

  a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

  b Select the linearization input/output set from the **Linearization I/O** area.

  If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

  If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Alternatively, you can use the Check Step Response Characteristics block to specify step response bounds for a signal.

See Also

"Design Optimization to Meet Step Response Requirements (GUI)"

Track Reference Signals

Use reference tracking to force a model signal to match a desired signal.

To track a reference signal:

1 In the Response Optimization tool, select **Signal Tracking** in the **New** drop-down list. A window opens where you specify the reference signal to track.

2 Specify a requirement name in the **Name** box.

3 Define the reference signal by entering vectors, or variables from the workspace, in the **Time vector** and **Amplitude** fields.
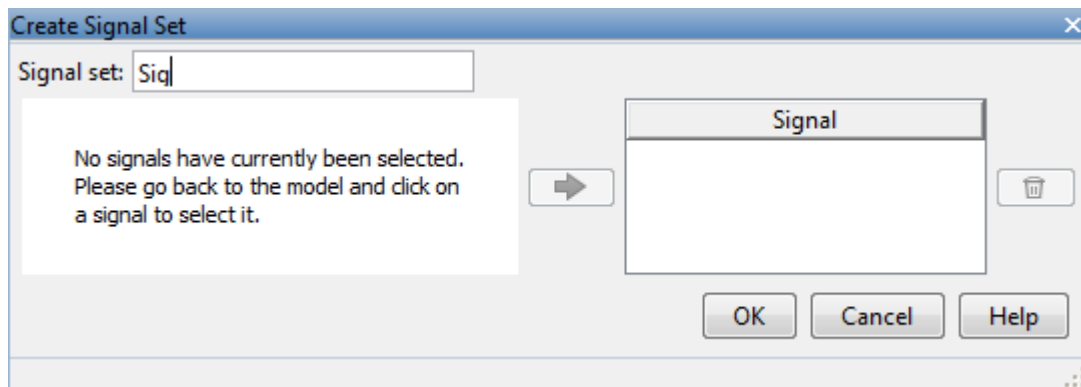
Click **Update reference signal data** to use the new amplitude and time vector as the reference signal.

4 Specify how the optimization solver minimizes the error between the reference and model signals using the **Tracking Method** list:

  • SSE — Reduces the sum of squared errors

  • SAE — Reduces the sum of absolute errors

5 In the **Specify Signal to Track Reference Signal** area, select a logged signal to apply the requirement to.

If you already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, they appear in the list. Select the corresponding check-box.

If you haven't selected a signal to log:

a   Click [+]. A window opens where you specify the logged signal.

b   In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c   Select the signal and click [⇨] to add it to the signal set.

d   In the **Signal set** box, enter a name for the selected signal set.

     Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

e   Select the check-box corresponding to the signal and click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the signal bound also appears in the Response Optimization tool window.

Note: When tracking a reference signal, the software ignores the maximally feasible solution option. For more information on this option, see "Selecting Optimization Termination Options" on page 3-75.

Alternatively, you can use the Check Against Reference block to specify a reference signal to track.

See Also

"Design Optimization to Track Reference Signal (GUI)"

Specify Custom Requirements

To specify custom requirements, such as minimizing system energy:

1 In the Response Optimization tool, select **Custom Requirement** in the **New** list. A window opens where you specify the custom requirement.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the name of the function that contains the custom requirement in the **Function** box. The field must be specified as a function handle using @. The function must be on the MATLAB path. Click to review or edit the function.

If the function does not exist, clicking opens a template MATLAB file. Use this file to implement the custom requirement. The default function name is `myCustomRequirement`.

5 (Optional) If you want to prevent the solver from considering specific parameter combinations, select the **Error if constraint is violated** check box. Use this option for parameter-only constraints.

During an optimization iteration, the solver evaluates requirements with this option selected first.

- If the constraint is violated, the solver skips evaluating any remaining requirements and proceeds to the next iterate.

- If the constraint is *not* violated, the solver evaluates the remaining requirements for the current iterate. If any of the remaining requirements bound signals or systems, then the solver simulates the model .

For more information, see "Skip Model Simulation Based on Parameter Constraint Violation (GUI)".

Note: If you select this check box, then do not specify signals or systems to bound. If you *do* specify signals or systems, then this check box is ignored.

---

6 (Optional) Specify the signal or system, or both, to be bound.

You can apply this requirement to model signals, or a linearization of your Simulink model (requires Simulink Control Design software), or both.

Click **Select Signals and Systems to Bound (Optional)** to view the signal and linearization I/O selection area.

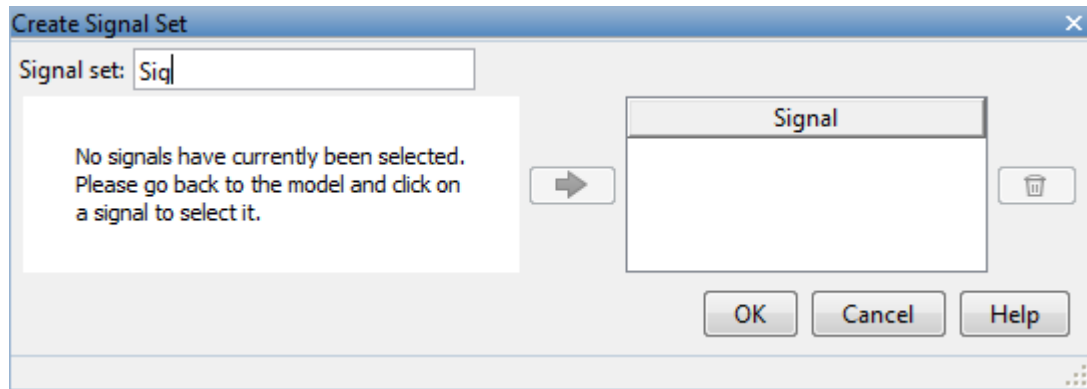- Apply this requirement to a model signal:

    In the **Signal** area, select a logged signal to which you will apply the requirement.

    If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check box.

    If you have not selected a signal to log:

    a Click [+]. A window opens where you specify the logged signal.

    b In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c   Select the signal and click to add it to the signal set.

d   In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

- Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a   Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b   Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box. For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.
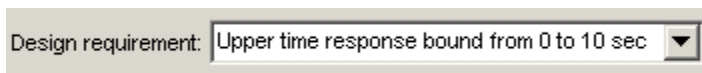
See Also

- "Design Optimization to Meet a Custom Objective (GUI)" on page 3-109
- "Design Optimization to Meet Custom Signal Requirements (GUI)" on page 3-136

Edit Design Requirements

The Edit Design Requirement dialog box allows you to exactly position constraint segments and to edit other properties of these constraints. The dialog box has two main components:

- An upper panel to specify the constraint you are editing
- A lower panel to edit the constraint parameters

The upper panel of the Edit Design Requirement dialog box resembles the image in the following figure.

In the context of the SISO Tool in Control System Toolbox software, **Design requirement** refers to both the particular editor within the SISO Tool that contains the requirement and the particular requirement within that editor. To edit other constraints within the SISO Tool, select another design requirement from the drop-down menu.

Edit Design Requirement Dialog Box Parameters

The particular parameters shown within the lower panel of the Edit Design Requirement dialog box depend on the type of constraint/requirement. In some cases, the lower panel contains a grid with one row for each segment and one column for each constraint parameter. The following table summarizes the various constraint parameters.

Edit Design Requirement Dialog Box Parameters

Parameter	Found in	Description
Time	Upper and lower time response bounds on step and impulse response plots	Defines the time range of a segment within a constraint/requirement.
Amplitude	Upper and lower time response bounds on step and impulse response plots	Defines the beginning and ending amplitude of a constraint segment.
Slope (1/s)	Upper and lower time response bounds	Defines the slope, in 1/s, of a constraint segment. It is an alternative method of specifying the magnitude values. Entering a new **Slope** value changes any previously defined magnitude values.
Final value	Step response bounds	Defines the input level after the step occurs.
Rise time	Step response bounds	Defines a constraint segment for a particular rise time.
% Rise	Step response bounds	The percentage of the step's range used to describe the rise time.
Settling time	Step response bounds	Defines a constraint segment for a particular settling time.
% Settling	Step response bounds	The percentage of the final value that defines the settling region used to describe the settling time.
% Overshoot	Step response bounds	
% Undershoot	Step response bounds	Defines the constraint segments for a particular percent undershoot.

Frequency-Domain Design Requirements in Simulink

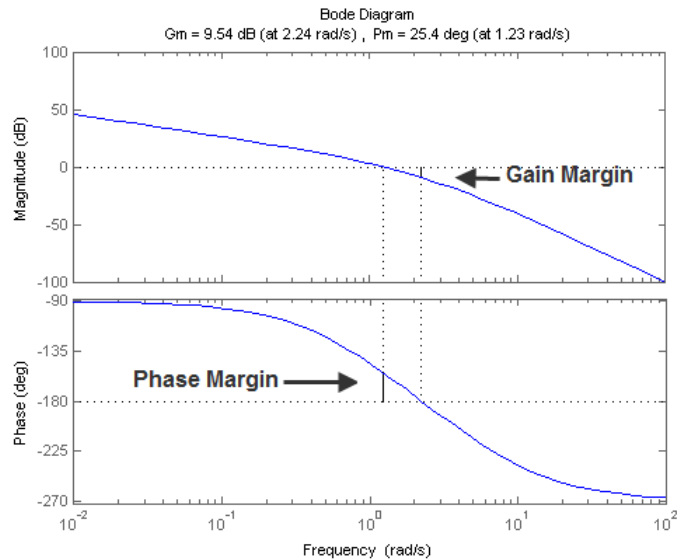| In this section... |
| --- |
| "Specify Lower Bounds on Gain and Phase Margin" on page 5-39 |
| "Specify Piecewise-Linear Lower and Upper Bounds on Frequency Response" on page 5-41 |
| "Specify Bound on Closed-Loop Peak Gain" on page 5-43 |
| "Specify Lower Bound on Damping Ratio" on page 5-45 |
| "Specify Upper and Lower Bounds on Natural Frequency" on page 5-47 |
| "Specify Upper Bound on Approximate Settling Time" on page 5-49 |
| "Specify Piecewise-Linear Upper and Lower Bounds on Singular Values" on page 5-51 |
| "Specify Step Response Characteristics" on page 5-13 |
| "Specify Custom Requirements" on page 5-32 |

Specify Lower Bounds on Gain and Phase Margin

To specify lower bounds on the gain and phase margin of a linear system:

1   In the Response Optimization tool, select **Gain and Phase Margin** in the **New** list. A window opens where you specify lower bounds on the gain and phase margin of your linear system.

2   Specify a requirement name in **Name**.

3   Specify bounds on the gain margin or phase margin, or both.

Bode Diagram
Gm = 9.54 dB (at 2.24 rad/s) , Pm = 25.4 deg (at 1.23 rad/s)

- **Gain margin** — Amount of gain increase or decrease required to make the loop gain unity at the frequency where the phase angle is –180°.

- **Phase margin** — Amount of phase increase or decrease required to make the phase angle –180° when the loop gain is 1.0

To specify a lower bound on the gain margin or phase margin, or both, select the corresponding check box and enter the lower bound value.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

· "Move Constraints Graphically"

· "Position Constraints Exactly"

Alternatively, you can use the Check Gain and Phase Margins block to specify bounds on the gain and phase margin. (Requires Simulink Control Design.)

Specify Piecewise-Linear Lower and Upper Bounds on Frequency Response

To specify upper or lower bounds on the magnitude of a system response:

1 In the Response Optimization tool, select **Bode Magnitude** in the **New** list. A window opens where you specify the lower or upper bounds on the magnitude of the system response.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the edge start and end frequencies and corresponding magnitude in the **Frequency** and **Magnitude** columns.

5 Insert or delete bound edges.

Click [+] to specify additional bound edges.

Select a row and click [🗑] to delete a bound edge.

6 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
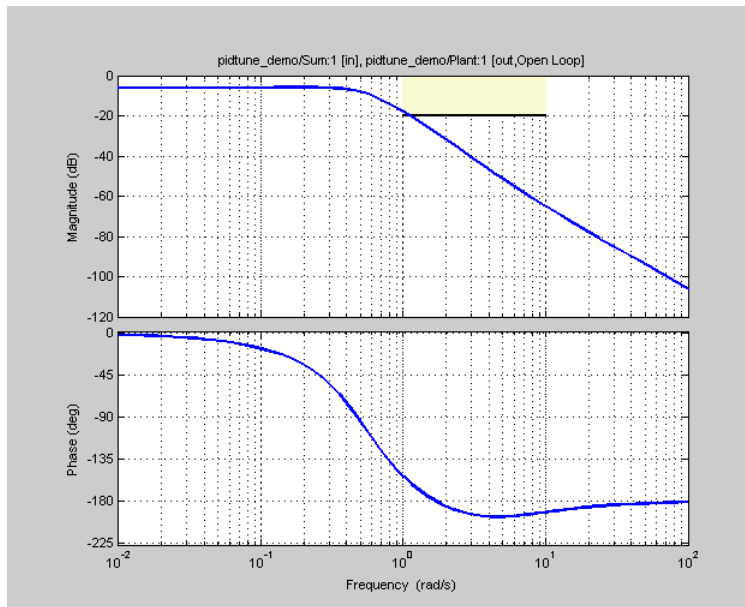
If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool window. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Bode Characteristics block to specify bounds on the magnitude of the system response. (Requires Simulink Control Design.)

Specify Bound on Closed-Loop Peak Gain

To specify an upper bound on the closed-loop peak response of a system:

1 In the Response Optimization tool, select **Closed-Loop Peak Gain** in the **New** list. A window opens where you specify an upper bound on the closed-loop peak gain of the system.

2 Specify a requirement name in the **Name** box.

3 Specify the upper bound on the closed-loop peak gain in the **Closed-Loop peak gain** box.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
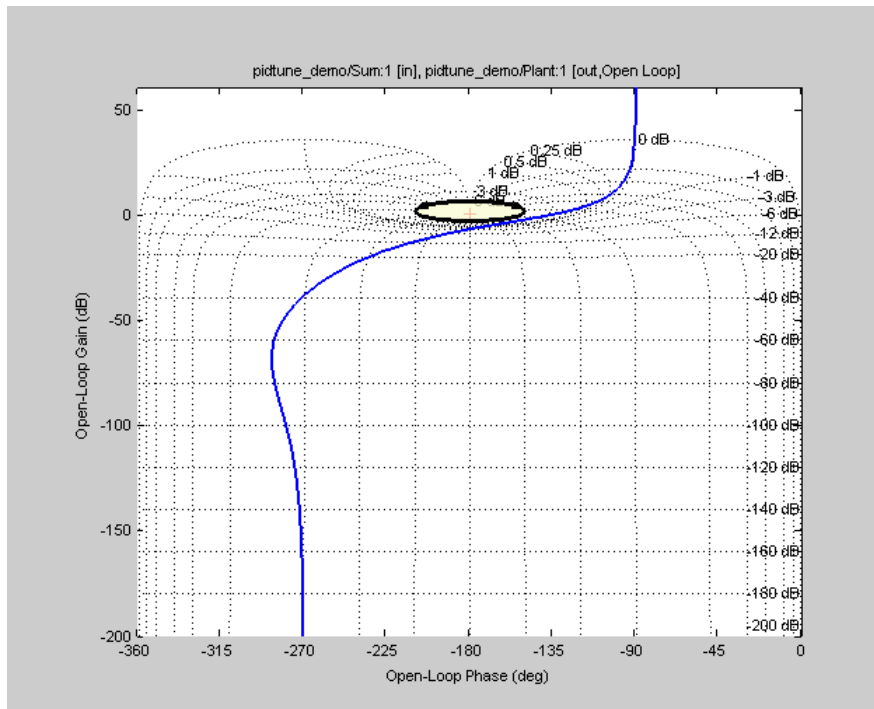
If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool window. A graphical display of the requirement also appears in the Response Optimization tool window.

6   (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Nichols Characteristics block to specify bounds on the magnitude of the system response. (Requires Simulink Control Design.)

Specify Lower Bound on Damping Ratio

To specify a lower bound on the damping ratio of the system:

1   In the Response Optimization tool, select **Damping Ratio** in the **New** list. A window opens where you specify an upper bound on the damping ratio of the system.
2   Specify a requirement name in the **Name** box.
3   Specify the lower bound on the damping ratio in the **Damping ratio** box.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
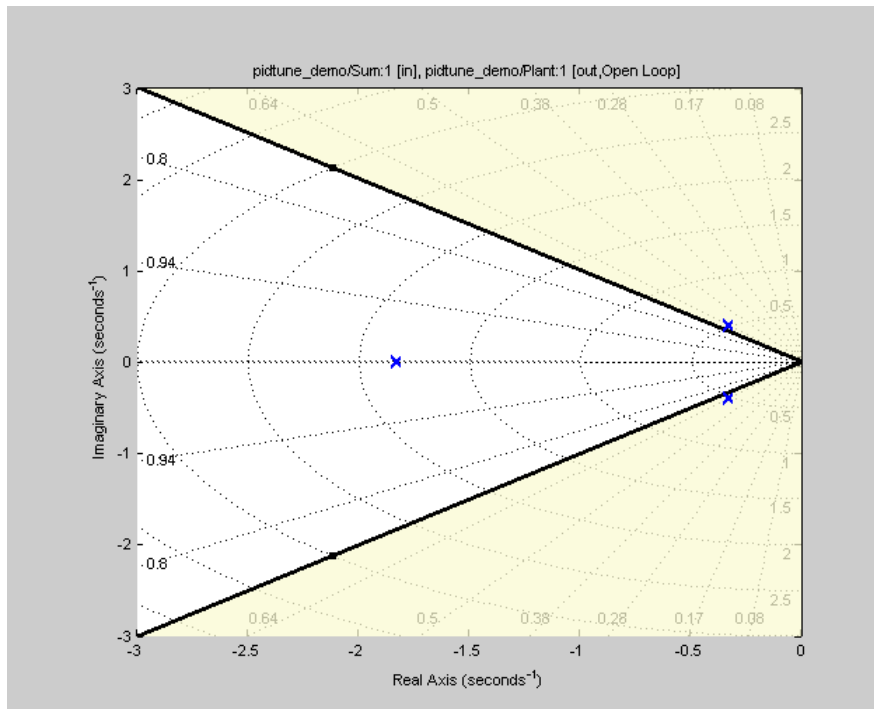
If you have not created a linearization input/output set, click $\boxed{+}$ to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify a bound on the damping ratio. (Requires Simulink Control Design.)

Specify Upper and Lower Bounds on Natural Frequency

To specify a bound on the natural frequency of the system:

1 In the Response Optimization tool, select **Natural Frequency** in the **New** list. A window opens where you specify a bound on the natural frequency of the system.

2 Specify a requirement name in the **Name** box.

3 Specify a lower or upper bound on the natural frequency in the **Natural frequency** box.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
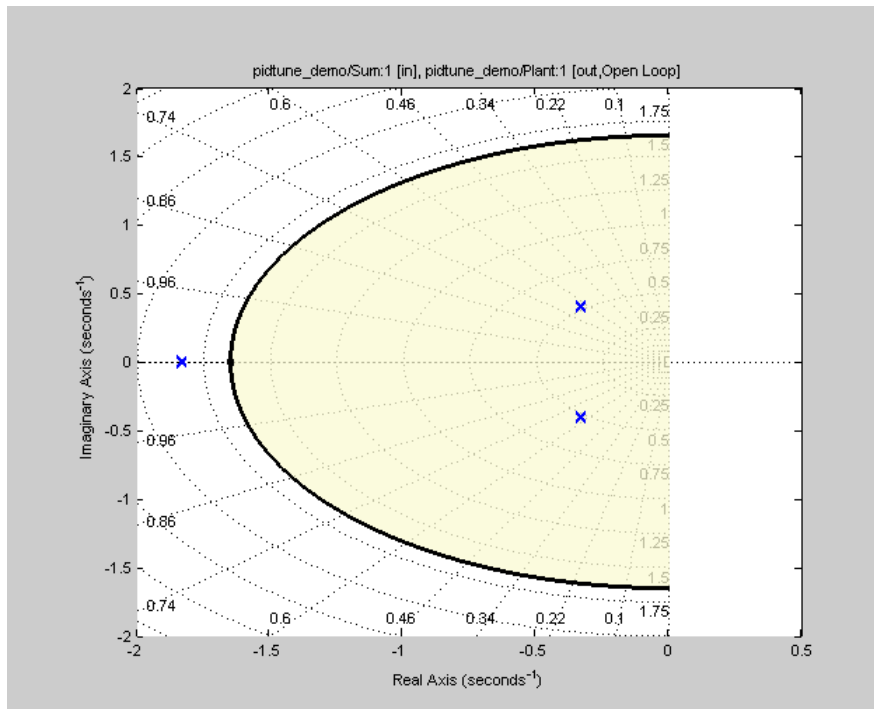
If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify a bound on the natural frequency. (Requires Simulink Control Design.)

Specify Upper Bound on Approximate Settling Time

To specify an upper bound on the approximate settling time of the system:

1 In the Response Optimization tool, select **Settling Time** in the **New** list. A window opens where you specify an upper bound on the approximate settling time of the system.

2 Specify a requirement name in the **Name** box.

3 Specify the upper bound on the approximate settling time in the **Settling time** box.

4 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
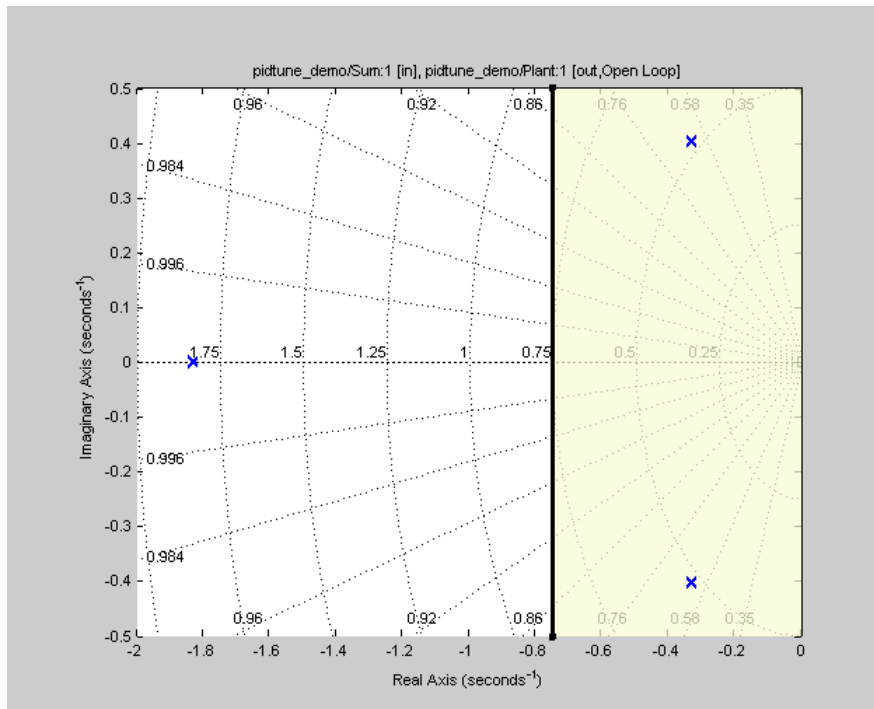
If you have not created a linearization input/output set, click ⊞ to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

6 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Pole-Zero Characteristics block to specify the approximate settling time. (Requires Simulink Control Design.)

Specify Piecewise-Linear Upper and Lower Bounds on Singular Values

To specify piecewise-linear upper and lower bounds on the singular values of a system:

1 In the Response Optimization tool, select **Singular Values** in the **New** list. A window opens where you specify the lower or upper bounds on the singular values of the system.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the edge start and end frequencies and corresponding magnitude in the **Frequency** and **Magnitude** columns, respectively.

5 Insert or delete bound edges.

Click [+] to specify additional bound edges.

Select a row and click [🗑] to delete a bound edge.

6 In the **Select Systems to Bound** section, select the linear systems to which this requirement applies.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.
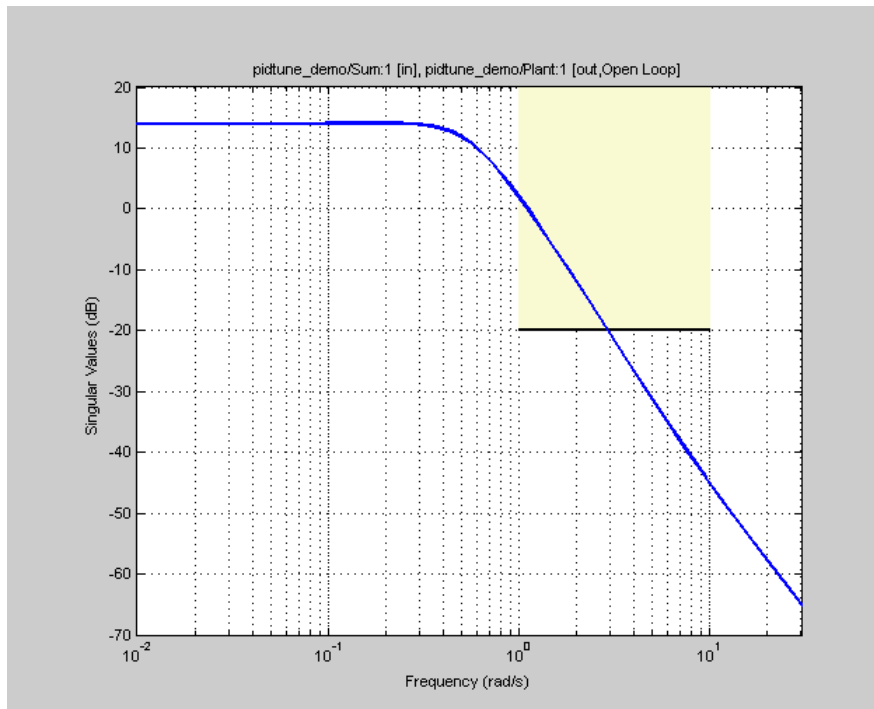
If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable with the specified name appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

8 (Optional) In the graphical display, you can:

- "Move Constraints Graphically"
- "Position Constraints Exactly"

Alternatively, you can use the Check Singular Value Characteristics block to specify bounds on the singular value. (Requires Simulink Control Design.)

Specify Step Response Characteristics
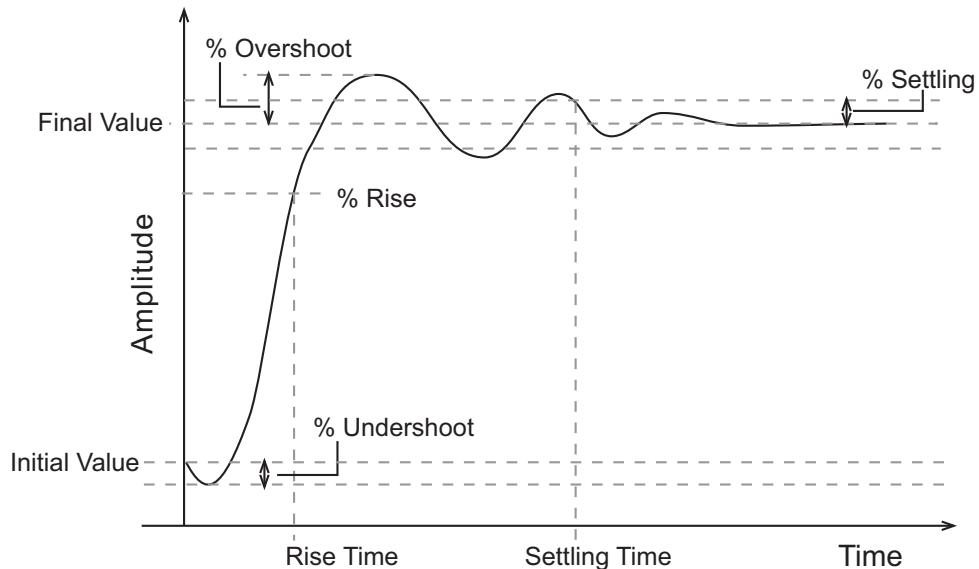
To specify step response characteristics:

1 You can apply this requirement to either a signal or a linearization of your model.

   In the Response Optimization Tool, click **New**. To apply this requirement to a signal, select the **Step Response Envelope** entry in the **New Time Domain**

Requirement section of the **New** list. To apply this requirement to a linearization of your model, select the **Step Response Envelope** entry in the **New Frequency Domain Requirement** section of the **New** list. The latter option requires Simulink Control Design software.

A window opens where you specify the step response requirements on a signal, or system.

2 Specify a requirement name in the **Name** box.

3 Specify the step response characteristics:

- **Initial value**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final value**: Input level after the step occurs
- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Rise**: The percentage used in the rise time.

- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.

- **% Settling**: The percentage used in the settling time.

- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

4   Specify the signals or systems to be bound.

You can apply this requirement to a model signal or to a linearization of your Simulink model (requires Simulink Control Design software).
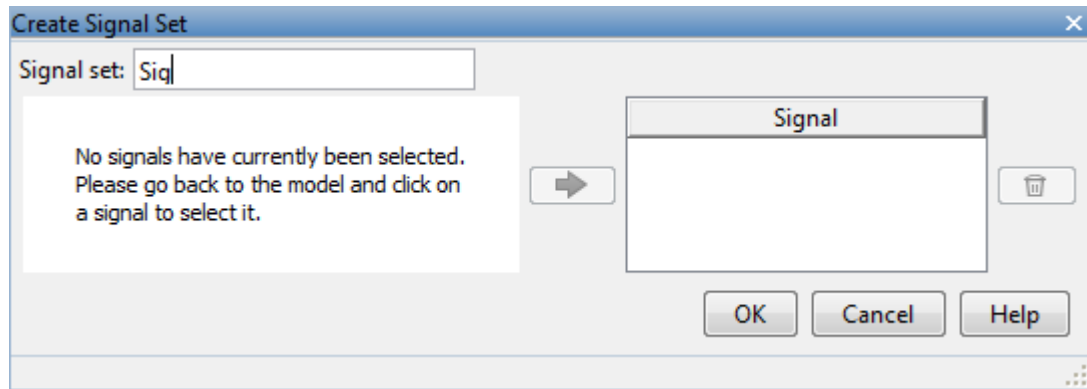
- Apply this requirement to a model signal:

    In the **Select Signals to Bound** area, select a logged signal to which you will apply the requirement.

    If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check-box.

    If you haven't selected a signal to log:

    a   Click [+]. A window opens where you specify the logged signal.

    b   In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click [→] to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

· Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click [+] to open the Create linearization I/O set dialog box.

For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

5 Click **OK**.

A variable with the specified requirement name appears in the **Data** area of the tool. A graphical display of the requirement also appears in the Response Optimization tool window.

Alternatively, you can use the Check Step Response Characteristics block to specify step response bounds for a signal.

See Also

"Design Optimization to Meet Step Response Requirements (GUI)"

Specify Custom Requirements

To specify custom requirements, such as minimizing system energy:

1 In the Response Optimization tool, select **Custom Requirement** in the **New** list. A window opens where you specify the custom requirement.

2 Specify a requirement name in the **Name** box.

3 Specify the requirement type using the **Type** list.

4 Specify the name of the function that contains the custom requirement in the **Function** box. The field must be specified as a function handle using @. The function must be on the MATLAB path. Click [folder icon] to review or edit the function.

If the function does not exist, clicking [folder icon] opens a template MATLAB file. Use this file to implement the custom requirement. The default function name is `myCustomRequirement`.

5 (Optional) If you want to prevent the solver from considering specific parameter combinations, select the **Error if constraint is violated** check box. Use this option for parameter-only constraints.

During an optimization iteration, the solver evaluates requirements with this option selected first.

- If the constraint is violated, the solver skips evaluating any remaining requirements and proceeds to the next iterate.

- If the constraint is *not* violated, the solver evaluates the remaining requirements for the current iterate. If any of the remaining requirements bound signals or systems, then the solver simulates the model .

For more information, see "Skip Model Simulation Based on Parameter Constraint Violation (GUI)".

Note: If you select this check box, then do not specify signals or systems to bound. If you *do* specify signals or systems, then this check box is ignored.

6   (Optional) Specify the signal or system, or both, to be bound.

You can apply this requirement to model signals, or a linearization of your Simulink model (requires Simulink Control Design software), or both.

Click **Select Signals and Systems to Bound (Optional)** to view the signal and linearization I/O selection area.
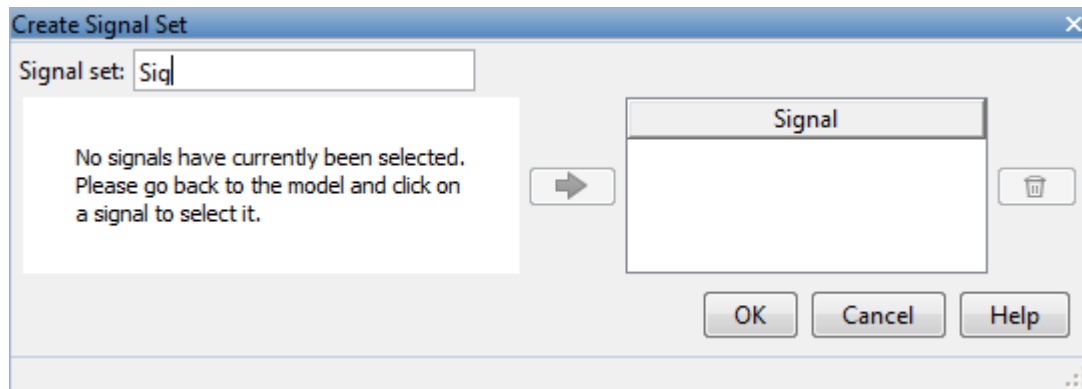
- Apply this requirement to a model signal:

  In the **Signal** area, select a logged signal to which you will apply the requirement.

  If you have already selected a signal to log, as described in "Specify Signals to Log" on page 3-12, it appears in the list. Select the corresponding check box.

  If you have not selected a signal to log:

  a   Click [+]. A window opens where you specify the logged signal.

  b   In the Simulink model window, click the signal to which you want to add a requirement.

The window updates and displays the name of the block and the port number where the selected signal is located.

c Select the signal and click to add it to the signal set.

d In the **Signal set** box, enter a name for the selected signal set.

Click **OK**. A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool window.

· Apply this requirement to a linear system.

Linear systems are defined by snapshot times at which the model is linearized and sets of linearization I/O points defining the system inputs and outputs.

a Specify the simulation time at which the model is linearized using the **Snapshot Times** box. For multiple simulation snapshot times, specify a vector.

b Select the linearization input/output set from the **Linearization I/O** area.

If you have already created a linearization input/output set, it will appear in the list. Select the corresponding check box.

If you have not created a linearization input/output set, click to open the Create linearization I/O set dialog box. For more information on using this dialog box, see "Create Linearization I/O Sets".

For more information on linearization, see "What Is Linearization?".

7 Click **OK**.

A new variable, with the specified name, appears in the **Data** area of the Response Optimization tool. A graphical display of the requirement also appears in the Response Optimization tool window.

See Also

- "Design Optimization to Meet a Custom Objective (GUI)" on page 3-109
- "Design Optimization to Meet Custom Signal Requirements (GUI)" on page 3-136

Time- and Frequency-Domain Requirements in SISO Design Tool

In this section...
"Root Locus Diagrams" on page 5-38
"Open-Loop and Prefilter Bode Diagrams" on page 5-40
"Open-Loop Nichols Plots" on page 5-40
"Step/Impulse Response Plots" on page 5-41

Root Locus Diagrams

- "Settling Time" on page 5-38
- "Percent Overshoot" on page 5-38
- "Damping Ratio" on page 5-39
- "Natural Frequency" on page 5-39
- "Region Constraint" on page 5-39

Settling Time

If you specify a settling time in the continuous-time root locus, a vertical line appears on the root locus plot at the pole locations associated with the value provided (using a first-order approximation). In the discrete-time case, the constraint is a curved line.

It is required that $\text{Re}\{pole\} < -4.6 / T_{settling}$ for continuous systems and

$\log(\text{abs}(pole)) / T_{discrete} < -4.6 / T_{settling}$ for discrete systems. This is an approximation of the settling time based on second-order dominant systems.

Percent Overshoot

Specifying percent overshoot in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the percent value (using a second-order approximation). In the discrete-time case, the constraint appears as two curves originating at (1,0) and meeting on the real axis in the left-hand plane.

The percent overshoot *p.o* constraint can be expressed in terms of the damping ratio, as in this equation:

$$p.o. = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$$

where ζ is the damping ratio.

Damping Ratio

Specifying a damping ratio in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the damping ratio. In the discrete-time case, the constraint appears as curved lines originating at (1,0) and meeting on the real axis in the left-hand plane.

The damping ratio defines a requirement on $-\mathrm{Re}\{pole\}/\mathrm{abs}(pole)$ for continuous systems and on

$$r = \mathrm{abs}(pSys)$$
$$t = \mathrm{angle}(pSys)$$
$$c = -\log(r)/\sqrt{(\log(r))^2 + t^2}$$

for discrete systems.

Natural Frequency

If you specify a natural frequency, a semicircle centered around the root locus origin appears. The radius equals the natural frequency.

The natural frequency defines a requirement on *abs(pole)* for continuous systems and on

$$r = \mathrm{abs}(pSys)$$
$$t = \mathrm{angle}(pSys)$$
$$c = \sqrt{(\log(r))^2 + t^2}/Ts_{model}$$

for discrete systems.

Region Constraint

Specifies an exclusion region in the complex plane, causing a line to appear between the two specified points with a shaded region below the line. The poles must not lie in the shaded region.

Open-Loop and Prefilter Bode Diagrams

Gain and Phase Margins

Specify a minimum phase and or a minimum gain margin.

Upper Gain Limit

You can specify an upper gain limit, which appears as a straight line on the Bode magnitude curve. You must select frequency limits, the upper gain limit in decibels, and the slope in dB/decade.

Lower Gain Limit

Specify the lower gain limit in the same fashion as the upper gain limit.

Open-Loop Nichols Plots

Phase Margin

Specify a minimum phase amount.

While displayed graphically at only one location around a multiple of -180 degrees, this requirement applies to phase margin regardless of actual phase (i.e., it is interpreted for all multiples of -180).

Gain Margin

Specify a minimum gain margin.

While displayed graphically at only one location around a multiple of -180 degrees, this requirement applies to gain margin regardless of actual phase (i.e., it is interpreted for all multiples of -180).

Closed-Loop Peak Gain

Specify a peak closed-loop gain at a given location. The specified value can be positive or negative in dB. The constraint follows the curves of the Nichols plot grid, so it is recommended that you have the grid on when using this feature.

While displayed graphically at only one location around a multiple of -180 degrees, this requirement applies to gain margin regardless of actual phase (i.e., it is interpreted for all multiples of -180).

Gain-Phase Requirement

Specifies an exclusion region for the response on the Nichols plot. The response must not pass through the shaded region.

This only applies to the region (phase and gain) drawn.

Step/Impulse Response Plots

- "Upper Time Response Bound" on page 5-41
- "Lower Time Response Bound" on page 5-41

Upper Time Response Bound

You can specify an upper time response bound.

Lower Time Response Bound

You can specify a lower time response bound.

Related Examples

- "How to Design Optimization-Based Controllers for LTI Systems" on page 5-43
- "Optimize LTI System to Meet Frequency-Domain Requirements" on page 5-44
- "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)"

Time-Domain Simulations in SISO Design Tool

When using a SISO Design Task, Simulink Design Optimization software automatically sets the model's simulation start and stop time and you cannot directly change them. By default, the simulation starts at 0 and continues until the SISO Design Task determines that the dynamics of the model have settled out. In addition, when the design requirements extend beyond this point, the simulation continues to the extent of the design requirements. Although you cannot directly adjust the start or stop time of the simulation, you can adjust the design requirements to extend further in time and thus force the simulation to continue to a certain point.

How to Design Optimization-Based Controllers for LTI Systems

To design optimization-based linear controller for an LTI model:

1 Create and import a linear model into a SISO Design Task. You can create an LTI model at the MATLAB command line, as described in "Creating an LTI Plant Model" on page 5-45.

2 Create a SISO Design Task with design and analysis plots, as described in "Creating Design and Analysis Plots" on page 5-46.

To learn more about SISO Design Tool, see "Using the SISO Design Task in the Controls & Estimation Tools Manager" in the Control System Toolbox documentation.

3 Under **Automated Tuning** select `Optimization based tuning` as the **Design Method** and then click the **Optimize Compensators** button to create a **Response Optimization** task within the Control and Estimation Tools Manager. See "Creating a Response Optimization Task" on page 5-48 for more information.

4 Within the **Response Optimization** node, select the **Compensators** pane to select and configure the compensator elements you want to tune during the response optimization. See "Selecting Tunable Compensator Elements" on page 5-50 for more information.

Note: Compensator elements or parameters cannot have uncertainty when used with frequency-domain based response optimization.

---

5 Under **Design requirements** in the **Response Optimization** node, select the design requirements you want the system to satisfy. See "Adding Design Requirements" on page 5-51 for more information.

6 Click the **Start Optimization** button within the **Response Optimization** node. The optimization progress results appear under **Optimization**. The **Compensators** pane contains the new, optimized compensator element values. See "Optimizing the System's Response" on page 5-59 for more information.

Optimize LTI System to Meet Frequency-Domain Requirements

Introduction

When you have Control System Toolbox software, you can place Simulink Design Optimization design requirements or constraints on plots in the SISO Design Tool graphical tuning editor and analysis plots that are part of a SISO Design Task. This allows you to include design requirements for response optimization in the frequency-domain in addition to the time-domain. This topic guides you through an example using frequency-domain design requirements to optimize the response of a system in the SISO Design Task.

You can specify frequency-domain design requirements to optimize response signals for any model that you can design within a SISO Design Task:

·   Command-line LTI models created with the Control System Toolbox commands

·   Simulink models that have been linearized using Simulink Control Design software

Design Requirements

In this example, you use a linearized version of the following Simulink model.
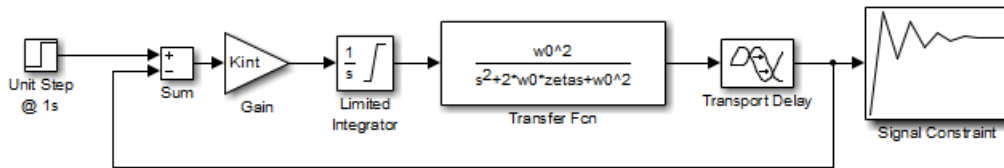
You use optimization methods to design a compensator so that the closed loop system meets the following design specifications when you excite the system with a unit step input:

- A maximum 30-second settling time
- A maximum 10% overshoot
- A maximum 10-second rise time
- A limit of ±0.7 on the actuator signal

Creating an LTI Plant Model

In the `srotut1` model, the plant model is composed of a gain, a limited integrator, a transfer function, and a transport delay.

You want to design the compensator for the open loop transfer function of the linearized `srotut1` model. The linearized `srotut1` plant model is composed of the gain, an unlimited integrator, the transfer function, and a Padé approximation to the transport delay.

To create an open loop transfer function based on the linearized `srotut1` model, enter the following commands:

```
w0      = 1;
zeta    = 1;
Kint    = 0.5;
Tdelay  = 1;
[delayNum,delayDen] = pade(Tdelay,1);
integrator    = tf(Kint,[1 0]);
transfer_fcn  = tf(w0^2,[1 2*w0*zeta w0^2]);
delay_block   = tf(delayNum,delayDen);
open_loopTF   = integrator*transfer_fcn*delay_block;
```

If the plant model is an array of models, the controller is designed for a nominal model only but you can analyze the control design for the remaining models in the array. For

more information, see "Control Design Analysis of Multiple Models" in the Control System Toolbox documentation.

**Tip** You can directly linearize the Simulink model using Simulink Control Design software.

Creating Design and Analysis Plots

This example uses a root locus diagram to design the response of the open loop transfer function, open_loopTF. To create a SISO Design Task, containing a root-locus plot for the open loop transfer function, use the following command:

```
controlSystemDesigner('rlocus',open_loopTF)
```

A **SISO Design Task** is created within the Control and Estimation Tools Manager, as shown in the following figure.
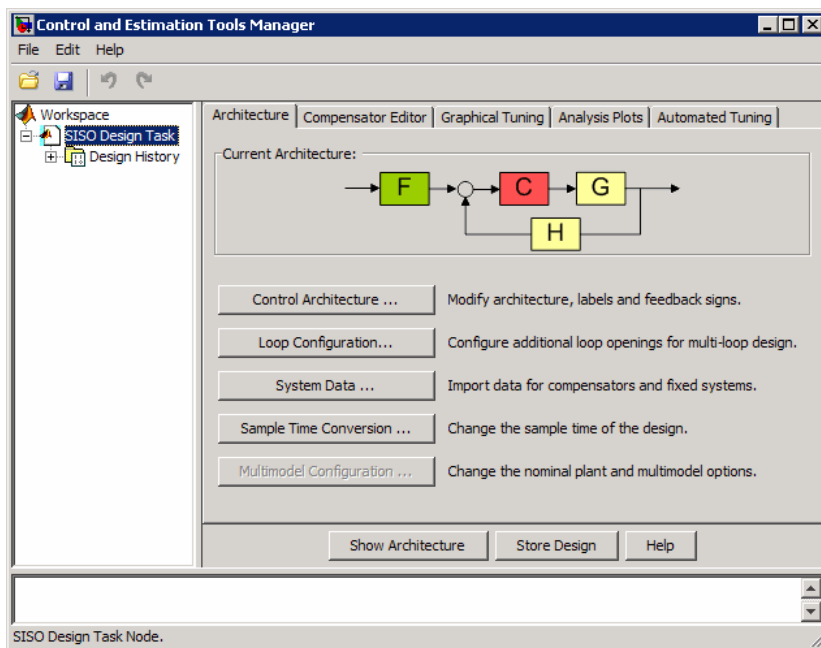
The Control and Estimation Tools Manager is a graphical environment for managing and performing tasks such as designing SISO systems. The SISO Design Task node contains five panels that perform actions related to designing SISO control systems. For more information, see "Using the SISO Design Task in the Controls & Estimation Tools Manager" in Control System Toolbox documentation.

The **Architecture** pane, within the **SISO Design Task** node, lets you choose the architecture for the control system you are designing. This example uses the default architecture. In this system, the plant model, *G*, is the open loop transfer function open_loopTF, the prefilter, *F*, and the sensor, *H*, are set to 1, and the compensator, *C*, is the compensator that will be designed using response optimization methods.

The SISO Design Task also contains a root locus diagram in the SISO Design Tool graphical tuning editor.
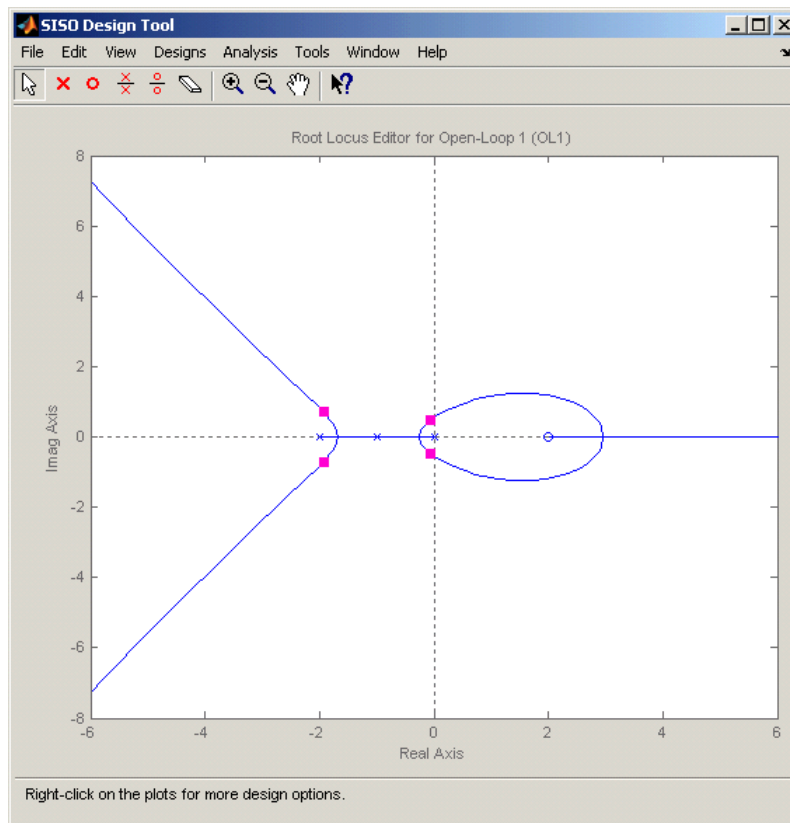
In addition to the root-locus diagram, it is helpful to visualize the response of the system with a step response plot. To add a step response:

1 Select the **Analysis Plots** pane with the **SISO Design Task** node of the Control and Estimation Tool Manager.

2 Select Step for the **Plot Type** of **Plot 1**.

3 Under **Contents of Plots**, select the check box in column **1** for the response **Closed Loop r to y**.

A step response plot appears in an Linear System Analyzer. The plot shows the response of the closed loop system from r (input to the prefilter, *F*) to y (output of the plant model, *G*):
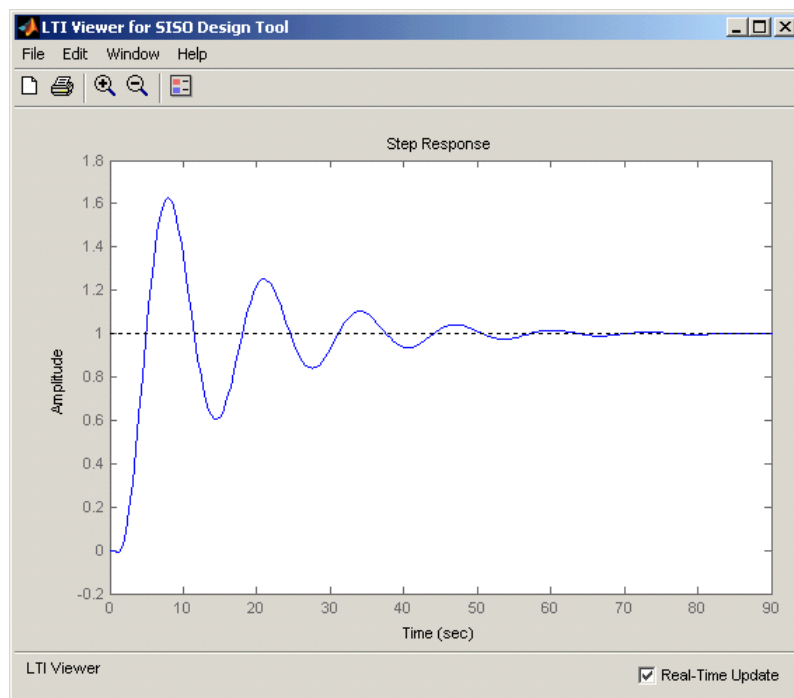
Creating a Response Optimization Task

There are several possible methods for designing a SISO system; this example uses an automated approach involving response optimization methods. After creating the design

and analysis plots as discussed in "Creating Design and Analysis Plots" on page 5-46, you are ready to start a response optimization task to design the compensator.

To create a response optimization task:

1   Select the **Automated Tuning** pane within the **SISO Design Task** node in the Control and Estimation Tools Manager.

2   In the **Automated Tuning** pane, select `Optimization based tuning` as the **Design Method**.

3   Click the **Optimize Compensators** button to create the **Response Optimization** node under the **SISO Design Task** node in the tree browser in the left pane of the Control and Estimation Tools Manager.

The **Response Optimization** node contains four panes as shown in the next figure.
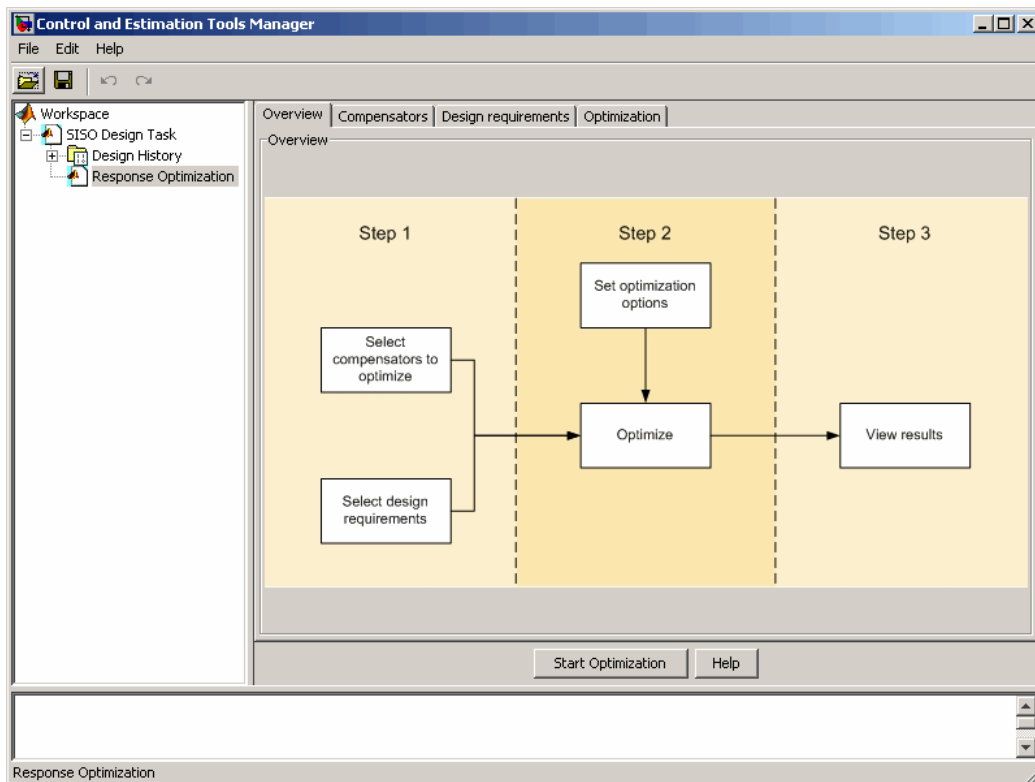
With the exception of the first pane, each corresponds to a step in the response optimization process:

- **Overview**: A schematic diagram of the response optimization process.

- **Compensators**: Select and configure the compensator elements that you want to tune. See "Selecting Tunable Compensator Elements" on page 5-50.

- **Design requirements**: Select the design requirements that you want the system to meet after tuning the compensator elements. See "Adding Design Requirements" on page 5-51.

- **Optimization**: Configure optimization options and view the progress of the response optimization. See "Optimizing the System's Response" on page 5-59.

Note: When optimizing responses in a SISO Design Task, you cannot add uncertainty to parameters or compensator elements.

Selecting Tunable Compensator Elements

You can tune elements or parameters within compensators in your system so that the response of the system meets the design requirements you specify. To specify the compensator elements to tune:

1 Select the **Compensators** pane within the **Response Optimization** node.

2 Within the **Compensators** pane, select the check boxes in the **Optimize** column that correspond to the compensator elements you want to tune.

In this example, to tune the **Gain** in the compensator **C**, select the check box next to this element, as shown in the following figure.
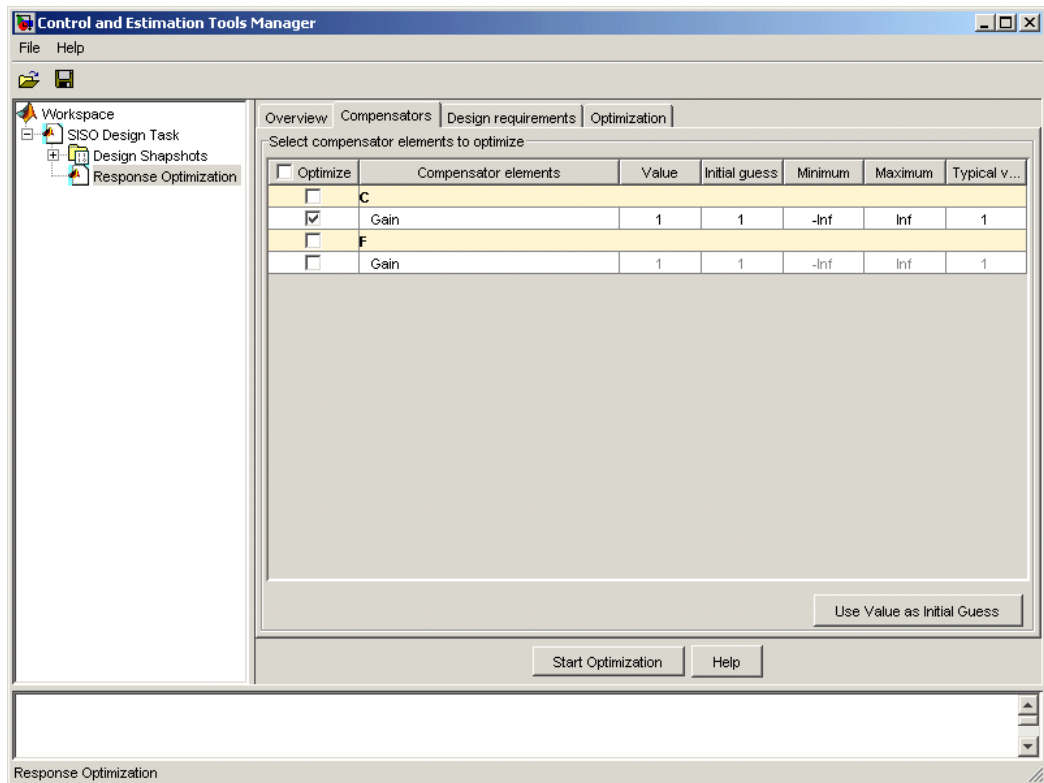
Note: Compensator elements or parameters cannot have uncertainty when used with frequency-domain based response optimization.

Adding Design Requirements

You can use both frequency-domain and time-domain design requirements to tune parameters in a control system. The **Design requirements** pane within the **Response Optimization** node of the Control and Estimation Tools Manager provides an interface to create new design requirements and select those you want to use for a response optimization.

This example uses the design specifications described in "Design Requirements" on page 5-44. The following sections each create a new design requirement to meet these specifications:

- "Settling Time Design Requirement" on page 5-52
- "Overshoot Design Requirement" on page 5-53
- "Rise Time Design Requirement" on page 5-54
- "Actuator Limit Design Requirement" on page 5-56

After you add the design requirements, you can select a subset of requirements for controller design, as described in "Selecting the Design Requirements to Use During Response Optimization" on page 5-59.

Settling Time Design Requirement

The first design specification for this example is to have a settling time of 30 seconds or less. This specification can be represented on a root-locus diagram as a constraint on the real parts of the poles of the open loop system.

To add this design requirement:

1 Select the **Design requirements** pane within the **Response Optimization** node of the Control and Estimation Tools Manager.

2 Click the **Add new design requirement** button. This opens the New Design Requirement dialog box.

Within this dialog box you can specify new design requirements and add them to a new or existing design or analysis plot.

3 Add a design requirement to the existing root-locus diagram:

a Select `Pole/zero settling time` from the **Design requirement type** menu.

b Select `Open-Loop L` from the **Requirement for response** menu.

c Enter 30 seconds for the **Settling time**.

d Click **OK**.

A vertical line should appear on the root-locus diagram, as shown in the following figure.
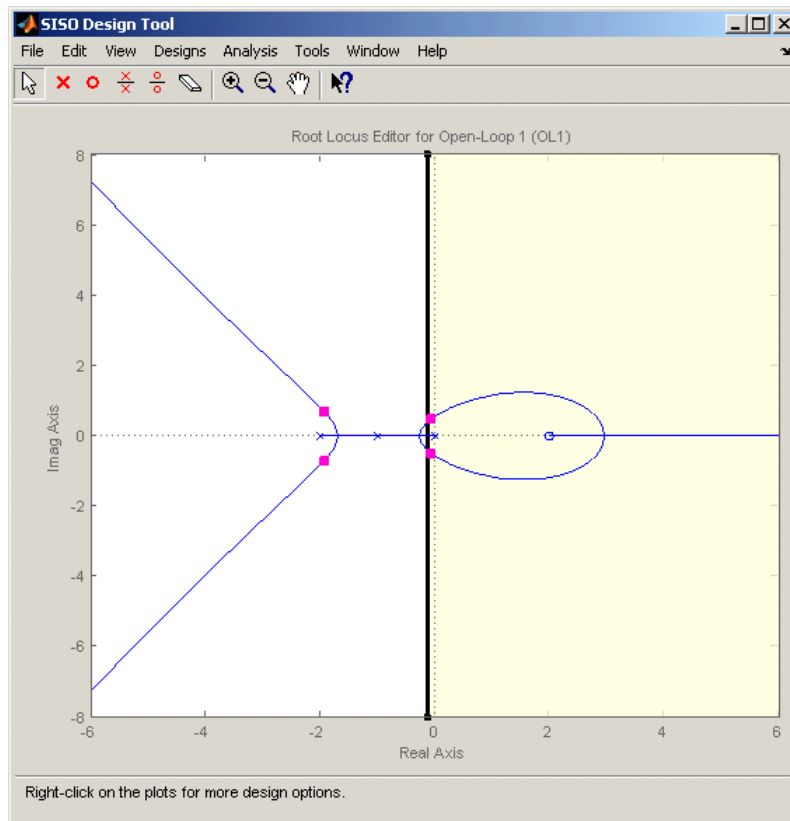
Overshoot Design Requirement

The second design specification for this example is to have a percentage overshoot of 10% or less. This specification is related to the damping ratio on a root-locus diagram. In addition to adding a design requirement with the **Add new design requirement** button, you can also right-click directly on the design or analysis plots to add the requirement, as shown next.

To add this design requirement:

1   Right-click anywhere within the white space of the root-locus diagram in the SISO Design Tool window. Select **Design Requirements** > **New** to open the New Design Requirement dialog box.

2   Select `Percent overshoot` as the **Design requirement type** and enter 10 as the
    Percent overshoot.

3   Click **OK** to add the design requirement to the root-locus diagram. The design
    requirement appears as two lines radiating at an angle from the origin, as shown in
    the following figure.
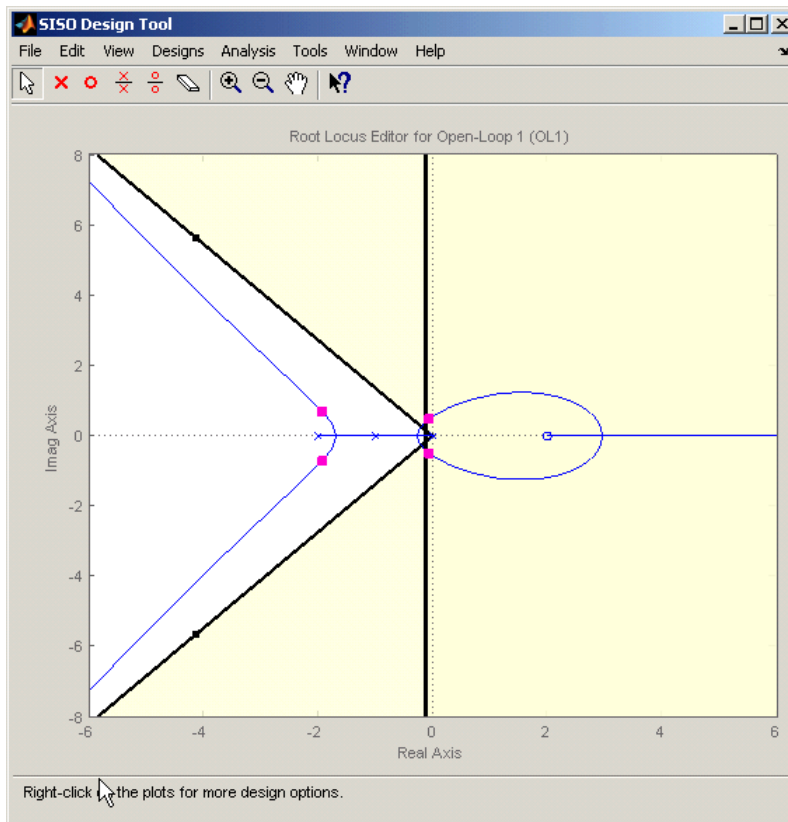
Rise Time Design Requirement

The third design specification for this example is to have a rise time of 10 seconds or less.
This specification is related to a lower limit on a Bode Magnitude diagram.

To add this design requirement:

1 Select the **Graphical Tuning** pane in the **SISO Design Task** node of the Control and Estimation Tools Manager.

2 For Plot 2, set **Plot Type** to `Open-Loop Bode`.

3 Right-click anywhere within the white space of the open-loop bode diagram in the SISO Design Tool window. Select **Design Requirements** > **New** to open the New Design Requirement dialog box.

4 Create a design requirement to represent the rise time and add it to the new Bode plot:

    a Select `Lower gain limit` from the **Design requirement type** menu.

    b Enter `1e-2` to `0.17` for the **Frequency** range.

    c Enter `0` to `0` for the **Magnitude** range.

    d Click **OK**.

A Bode diagram appears within the SISO Design Tool window. The magnitude plot of the Bode diagram includes a horizontal line representing the design requirement, as shown in the following figure.
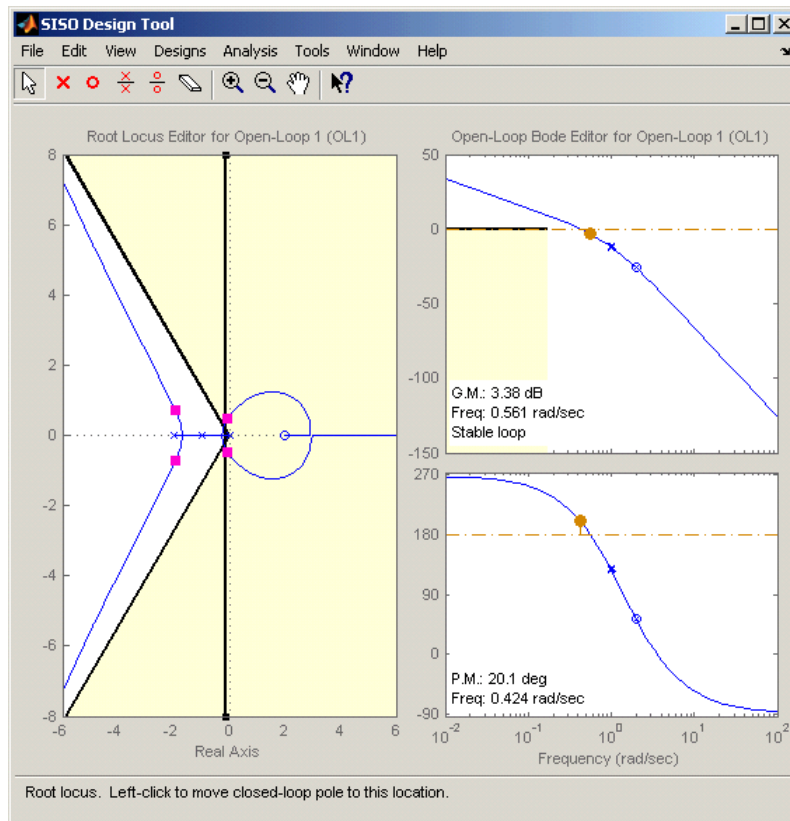
Actuator Limit Design Requirement

The fourth design specification for this example is to limit the actuator signal to within ±0.7. To add this design requirement:

1 Select the **Design requirements** pane in the **Response Optimization** node of the Control and Estimation Tools Manager.

2 Click the **Add new design requirement** button to open the New Design Requirement dialog box.

3 Create a time-domain design requirement to represent the upper limit on the actuator signal, and add it to a new step response plot in the Linear System Analyzer:

a Select `Step response upper amplitude limit` from the **Design requirement type** menu.

b Select `Closed Loop r to u` from the **Requirement for response** menu.

c Enter `0` to `10` for the **Time** range.

d Enter `0.7` to `0.7` for the **Amplitude** range.

e Click **OK**. A second step response plot for the closed loop response from `r` to `u` appears in the Linear System Analyzer. The plot contains a horizontal line representing the upper limit on the actuator signal.

f To extend this limit for all times (to $t = \infty$), right click on the black edge of the design requirement, somewhere toward the right edge, and select **Extend to inf**. The diagram should now appear as shown next.
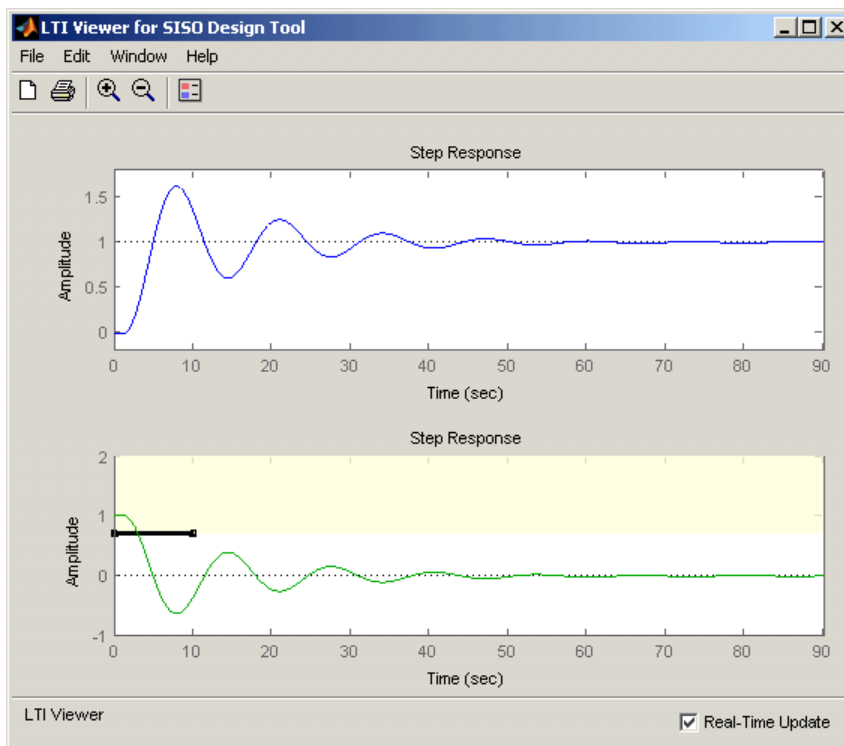
To add the corresponding design requirement for the lower limit on the actuator signal:

1 Select the **Design requirements** pane in the **Response Optimization** node of the Control and Estimation Tools Manager.

2 Click the **Add new design requirement** button to open the New Design Requirement dialog box.

3 Create a time-domain design requirement to represent the lower limit on the actuator signal, and add it to the step response plot in the Linear System Analyzer:

    a Select `Step response lower amplitude limit` from the **Design requirement type** menu.

    b Select `Closed Loop r to u` from the **Requirement for response** menu.

    c Enter `0` to `10` for the **Time** range.

    d Enter `-0.7` to `-0.7` for the **Amplitude** range.

    e Click **OK**. The step response plot now contains a second horizontal line representing the lower limit on the actuator signal.

    f To extend this limit for all times (to $t = \infty$), right-click in the yellow shaded area and select **Extend to inf**. The diagram should now appear as shown in the following figure.
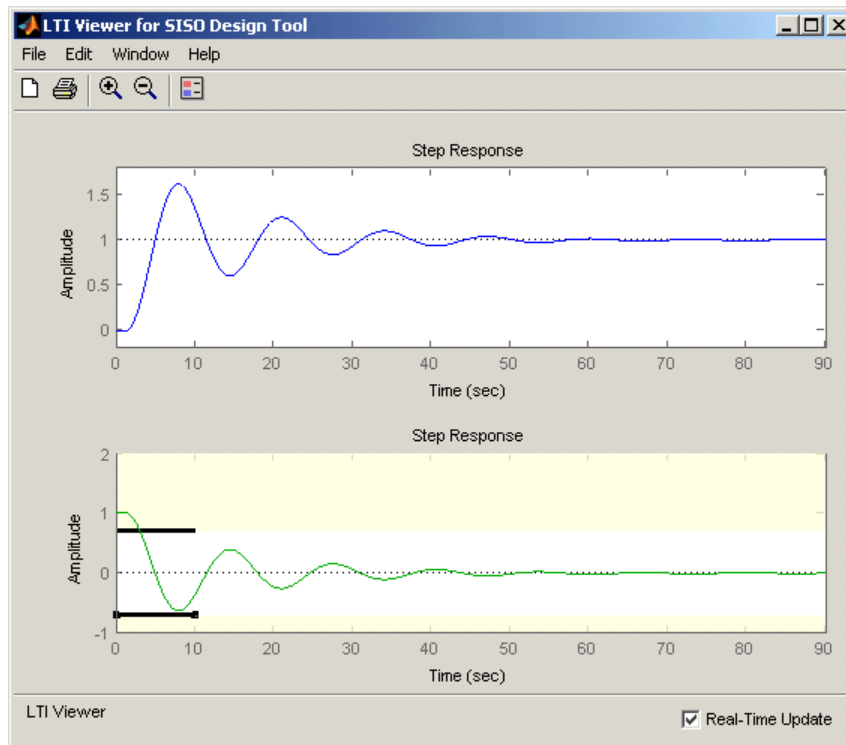
Selecting the Design Requirements to Use During Response Optimization

The design requirements give constraints on the dynamics of the system and the values of response signals. The table in the **Design requirements** tab lists all design requirements in the design and analysis plots. Select the check boxes next to the design requirements you want to use in the response optimization. This example uses all the current design requirements.

Optimizing the System's Response

After selecting the compensator elements to tune and adding design requirements for the response signals to satisfy, you are ready to being the response optimization.

The **Optimization** pane within the **Response Optimization** node of the Control and Estimation Tools Manager displays the progress of the response optimization. The pane

also contains options to configure the types of progress information displayed during the optimization and options to configure the optimization methods and algorithms.

To optimize the response of the system in this example, click the **Start Optimization** button.

The **Optimization** pane displays the progress of the optimization, iteration by iteration, as shown next. Termination messages from the optimization method and suggestions for improving convergence also appear here.
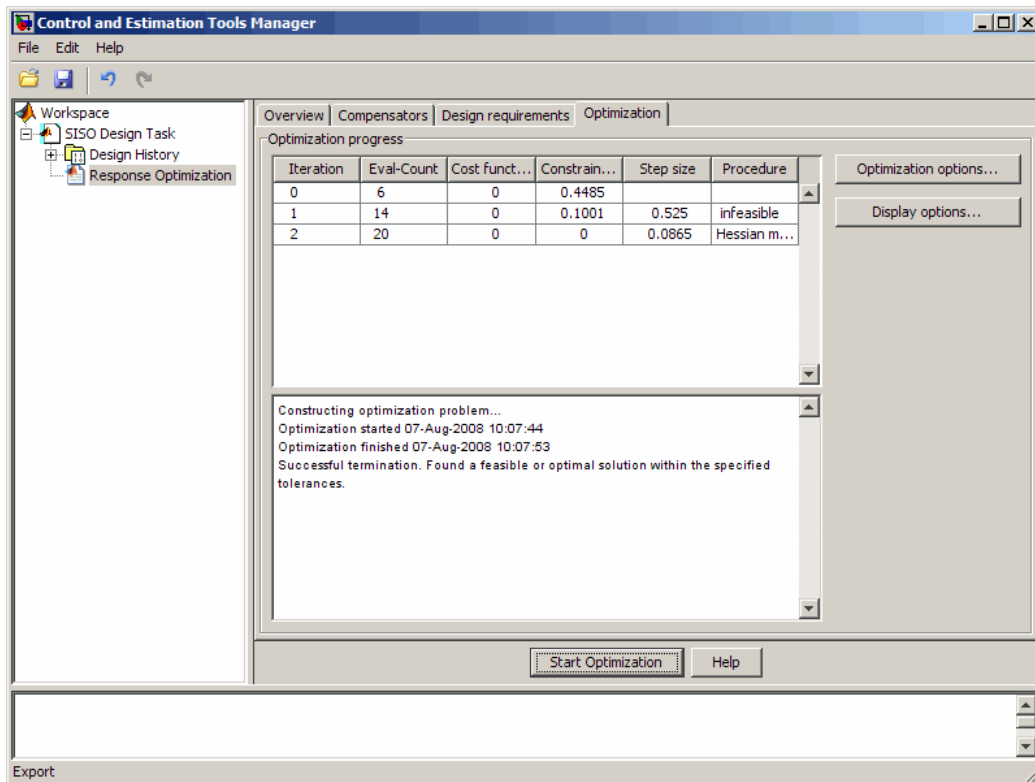
The optimized signals in the design and analysis plots appear as follows:
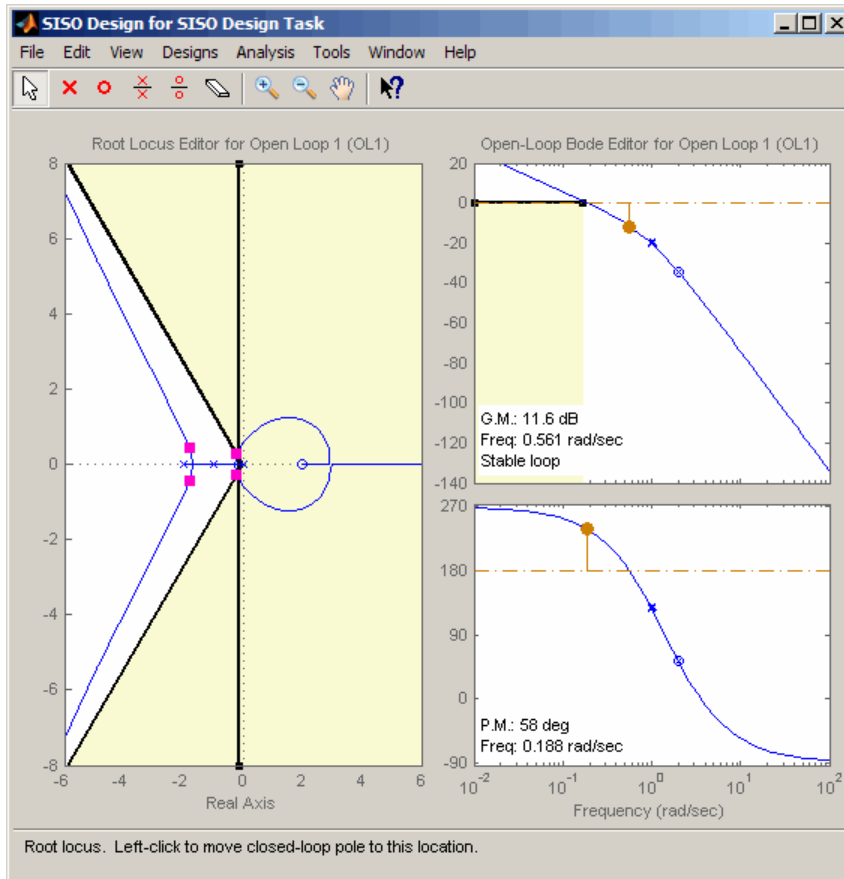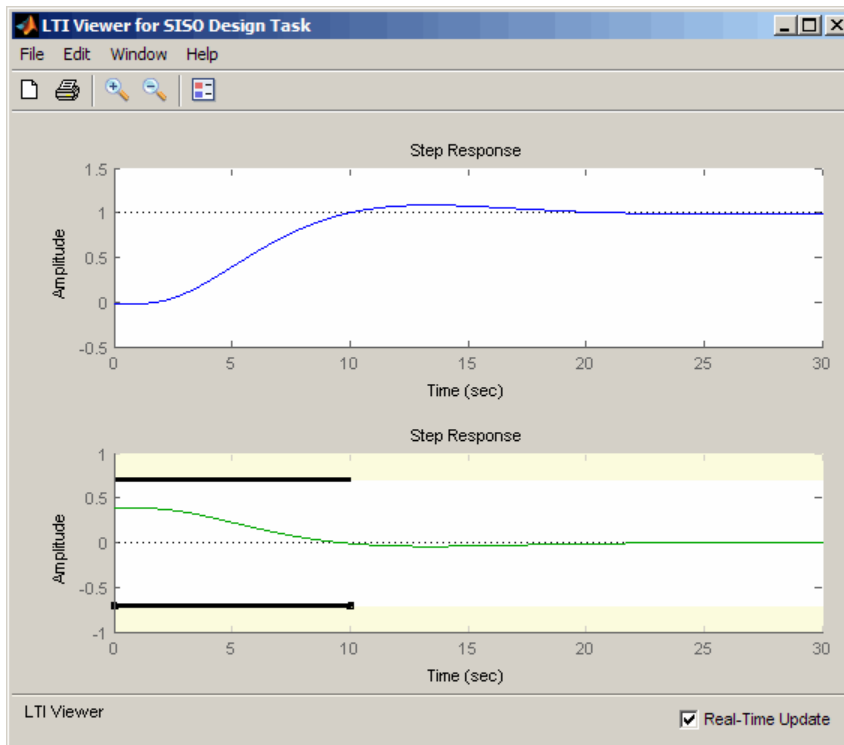
Creating and Displaying the Closed-Loop System

After designing a compensator by optimizing the response of the system, you can export the compensator to the MATLAB workspace, and create a model of the full closed-loop system.

1   Within the SISO Design Tool window, select **File** > **Export** to open the SISO Tool Export dialog box.

2   Select the compensator you designed, **Compensator C**, and then click the **Export to Workspace** button.

At the command line, enter the following command to create the closed-loop system, CL, from the open-loop transfer function, open_loopTF, and the compensator, C:

```
CL=feedback(C*open_loopTF,1)
```

This returns the following model:

```
Zero/pole/gain from input to output "Output":
              -0.19414 (s-2)
---------------------------------------------
(s^2 + 0.409s + 0.1136) (s^2 + 3.591s + 3.418)
```

To create a step response plot of the closed loop system, enter the following command:

```
step(CL);
```

This produces the following figure:
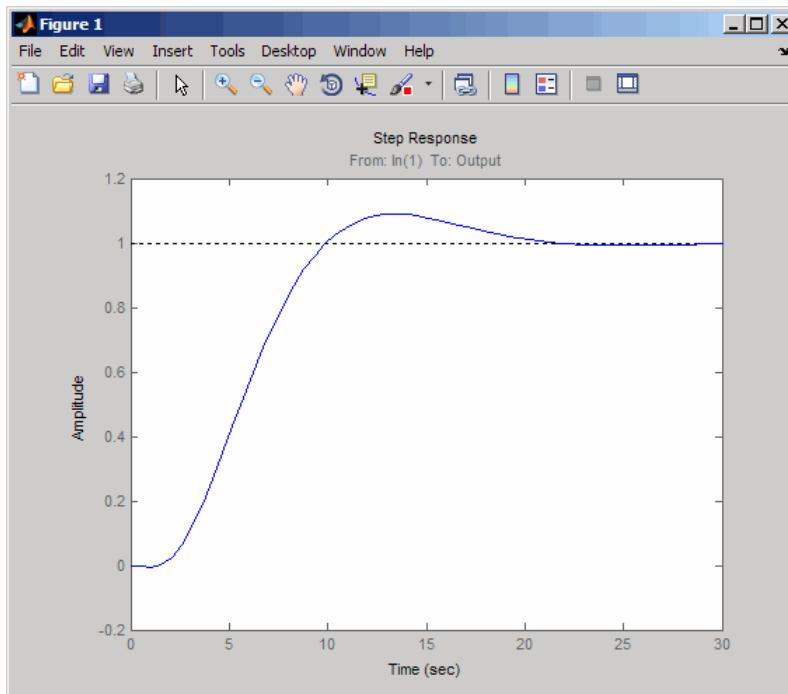
Designing Linear Controllers for Simulink Models

When you have Control System Toolbox and Simulink Control Design software, you can perform frequency-domain optimization of Simulink models.

You can use Simulink Control Design software to configure SISO Design Tool with compensators, inputs, outputs, and loops computed from a Simulink model. For more information, see "Creating a SISO Design Task" in Simulink Control Design documentation.

After you configure the SISO Design Tool, use Simulink Design Optimization software to optimize the controller parameters of the linearized Simulink model. For an example of optimization-based control design for a model linearized using Simulink Control Design software, see "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)".

There is only one difference when tuning compensators derived from Simulink Control Design software: The tuning of compensators from a Simulink model is done through the masks of the Simulink blocks representing each compensator. When selecting parameters to optimize, users can tune the compensator in the pole, zero, or gain format, or in a format consistent with the Simulink block mask as shown in the following figure. Changing the compensator format is not possible when optimizing pure SISO Tool models (those not derived using Simulink Control Design software).
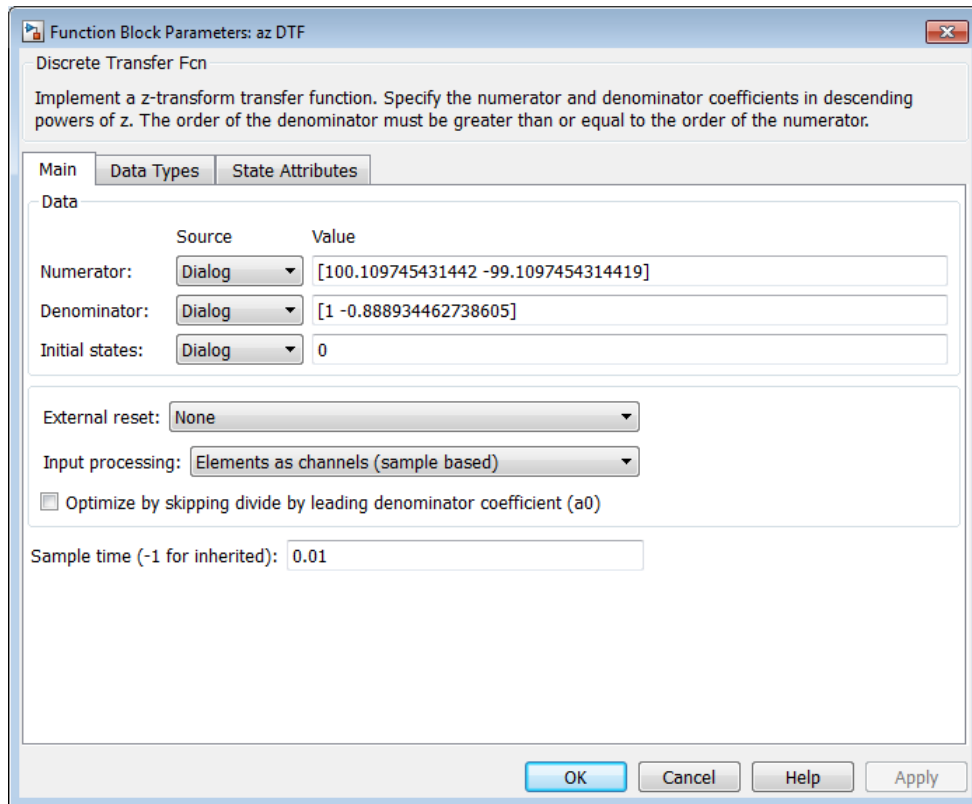
Mask of a Simulink compensator block
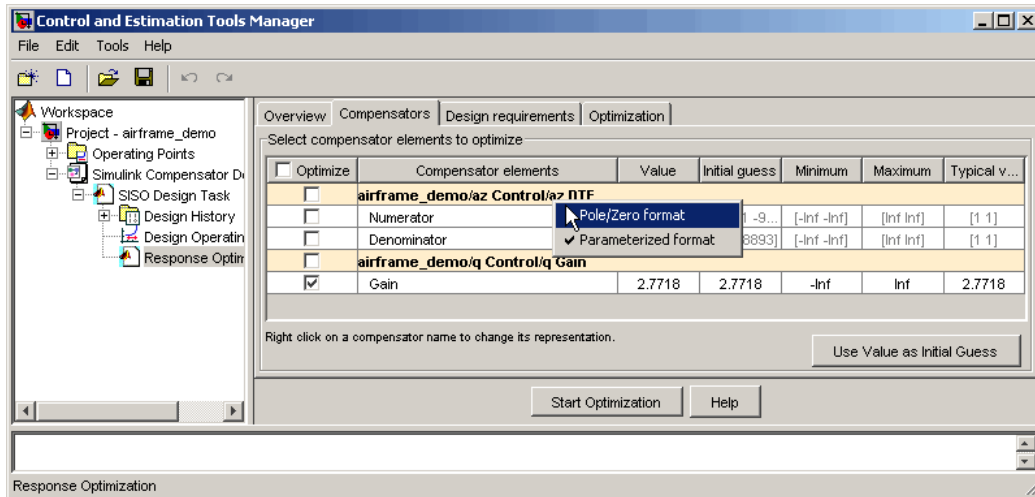
Response optimization compensators pane

6

Lookup Tables

What are Adaptive Lookup Tables?

Lookup Tables

Lookup tables store numeric data in a multidimensional array format. In the simpler two-dimensional case, lookup tables can be represented by matrices. Each element of a matrix is a numerical quantity, which can be precisely located in terms of two indexing variables. At higher dimensions, lookup tables can be represented by multidimensional matrices, whose elements are described in terms of a corresponding number of *indexing variables*.

Lookup tables provide a means to capture the dynamic behavior of a physical (mechanical, electronic, software) system. The behavior of a system with M inputs and N outputs can be approximately described by using N lookup tables, each consisting of an array with M dimensions.

You usually generate lookup tables by experimentally collecting or artificially creating the input and output data of a system. In general, you need as many indexing parameters as the number of input variables. Each indexing parameter may take a value within a predetermined set of data points, which are called the *breakpoints*. The set of all breakpoints corresponding to an indexing variable is called a *grid*. Thus, a system with M inputs is gridded by M sets of breakpoints. The software uses the breakpoints to locate the array elements, where the output data of the system are stored. For a system with N outputs, the software locates the N array elements and then stores the corresponding data at these locations.

After you create a lookup table using the input and output measurements as described previously, you can use the corresponding multidimensional array of values in applications without having to remeasure the system outputs. In fact, you need only the input data to locate the appropriate array elements in the lookup table because the software reads the approximate system output from the data stored at these locations. Therefore, a lookup table provides a suitable means of capturing the input-output mapping of a *static* system in the form of numeric data stored at predetermined array locations.

Adaptive Lookup Tables

Statically defined lookup tables, as described in "Lookup Tables" on page 6-2, cannot accommodate the *time-varying* behavior (characteristics) of a physical plant. Static lookup tables establish a permanent and static mapping of input-output behavior of a

physical system. Conversely, the behavior of actual physical systems often varies with time due to wear, environmental conditions, and manufacturing tolerances. With such variations, the static mapping of input-output behavior of a plant described by the lookup table may no longer provide a valid representation of the plant characteristics.

Adaptive lookup tables incorporate the time-varying behavior of physical plants into the lookup table generation and maintenance process while providing all of the functionality of a regular lookup table.

The adaptive lookup table receives the input and output measurements of a plant's behavior, which are then used to dynamically create and update the content of the underlying lookup table. In addition to requiring the input data to create the lookup table, the adaptive lookup table also uses the output data of the plant to recalculate the table values. For example, you can collect the output data of the plant by placing sensors at appropriate locations in a physical system.

The software uses the input measurements to locate the array elements by comparing these input values with the breakpoints defined for each indexing variable. Next, it uses the output measurements to recalculate the numeric value stored at these array locations. However, unlike a regular table, which only stores the array data before the actual use of the lookup table, the adaptive table continuously improves the content of the lookup table. This continuous improvement of the table data is referred to as the *adaptation process* or *learning process*.

The adaptation process involves statistical and signal processing algorithms to recapture the input-output behavior of the plant. The adaptive lookup table always tries to provide a valid representation of the plant dynamics even though the plant behavior may be time varying. The underlying signal processing algorithms are also robust against reasonable measurement noise and they provide appropriate filtering of noisy output measurements.

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

Related Examples
- "Model Engine Using n-D Adaptive Lookup Table" on page 6-44

More About
- "About Lookup Table Blocks"

- "Building Models Using Adaptive Lookup Table Blocks" on page 6-38

How to Estimate Lookup Table Values

You can use lookup table Simulink blocks to approximate a system's behavior, as described in "About Lookup Table Blocks" in the Simulink documentation. After you build your system using lookup tables, you can use Simulink Design Optimization software to estimate the table values from measured I/O data.

Estimating lookup table values is an example of estimating parameters which are matrices or multi-dimensional arrays. The workflow for estimating parameters of a lookup table consist of the following tasks:

1 Creating a Simulink model using lookup table blocks.
2 Importing the measured input and output (I/O) data from which you want to estimate the table values.
3 Analyzing and preparing the I/O data for estimation.
4 Estimating the lookup table values.
5 Validating the estimated table values using a validation data set.

Related Examples
- "Estimate Lookup Table Values from Data" on page 6-23
- "Estimate Constrained Values of a Lookup Table" on page 6-6

Estimate Constrained Values of a Lookup Table

In this section...
"Objectives" on page 6-6
"About the Data" on page 6-6
"Open a Parameter Estimation Session" on page 6-6
"Estimate the Monotonically Increasing Table Values Using Default Settings" on page 6-9
"Validate the Estimation Results" on page 6-17

Objectives

This example shows how to estimate constrained values of a lookup table. Apply monotonically increasing constraints to the lookup table values, and use the Parameter Estimation tool to estimate the table values.

About the Data

In this example, use `lookup_increasing.mat`, which contains the measured I/O data for estimating the lookup table values. The MAT-file includes the following variables:

- `xdata1` — Consists of 602 uniformly-sampled input data points in the range [-5,5].
- `ydata1` — Output data corresponding to the input data samples.

> **Note:** The output data is a monotonically increasing function of the input data.

- `time1` — Time vector.

Use the I/O data to estimate the values of the lookup table in the `lookup_increasing` Simulink model. The table contains eleven values, which are stored in the MATLAB variable `table`. To learn more about how to specify the table's values, see "Enter Breakpoints and Table Data" in the Simulink documentation.

Open a Parameter Estimation Session

To estimate the monotonically increasing lookup table values, first open a new parameter estimation session.

1   Open the lookup table model by typing the following command at the MATLAB
    prompt:

    ```
    lookup_increasing
    ```

    This command opens the Simulink model, and loads the estimation data in the
   MATLAB workspace.
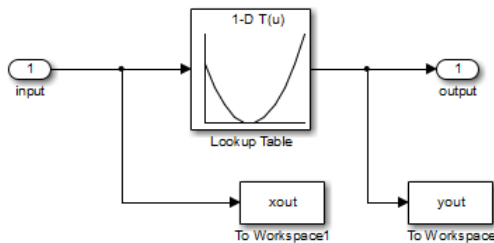
2   Double-click the Lookup Table block to view the monotonically increasing constraint
    applied to the table output values.
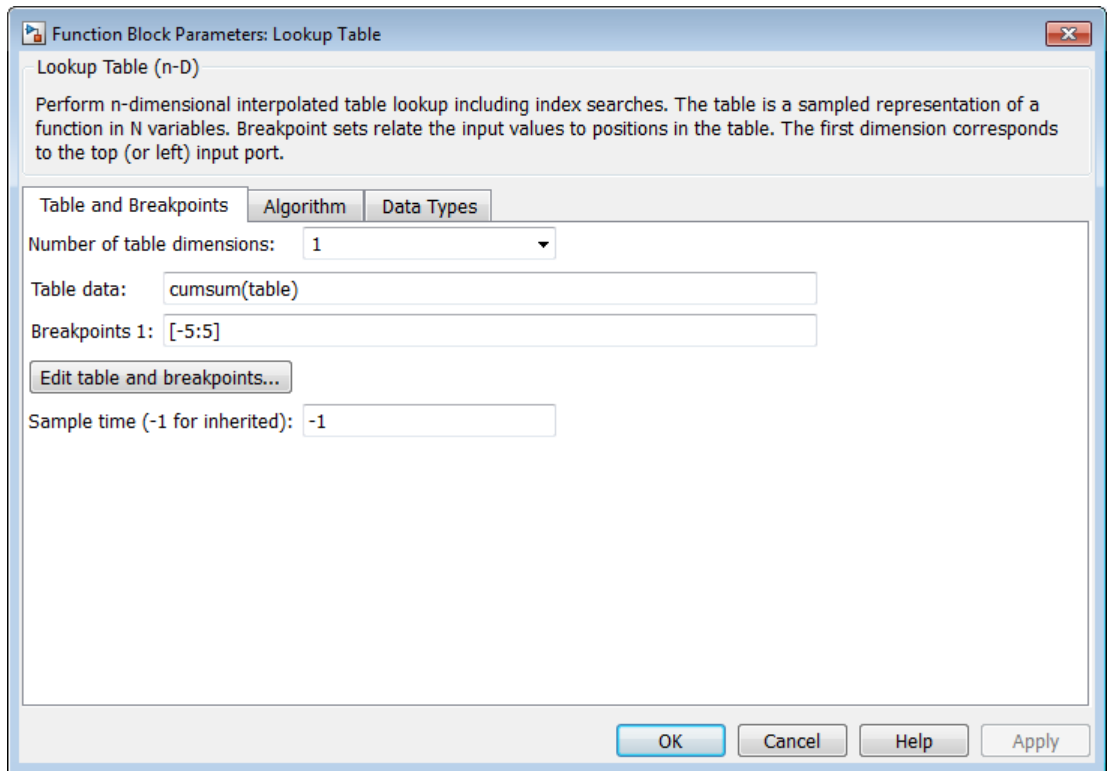
The **Table data** field of the Function Block Parameters dialog box shows the constraint. The cumulative sum function, cumsum, applies a monotonically increasing constraint on the table output values. This function computes the cumulative sum of the table values based on estimation of the individual table elements from the I/O data.

3 In the Simulink model, select **Analysis > Parameter Estimation** to open a session with the name **lookup_increasing** in the Parameter Estimation tool.
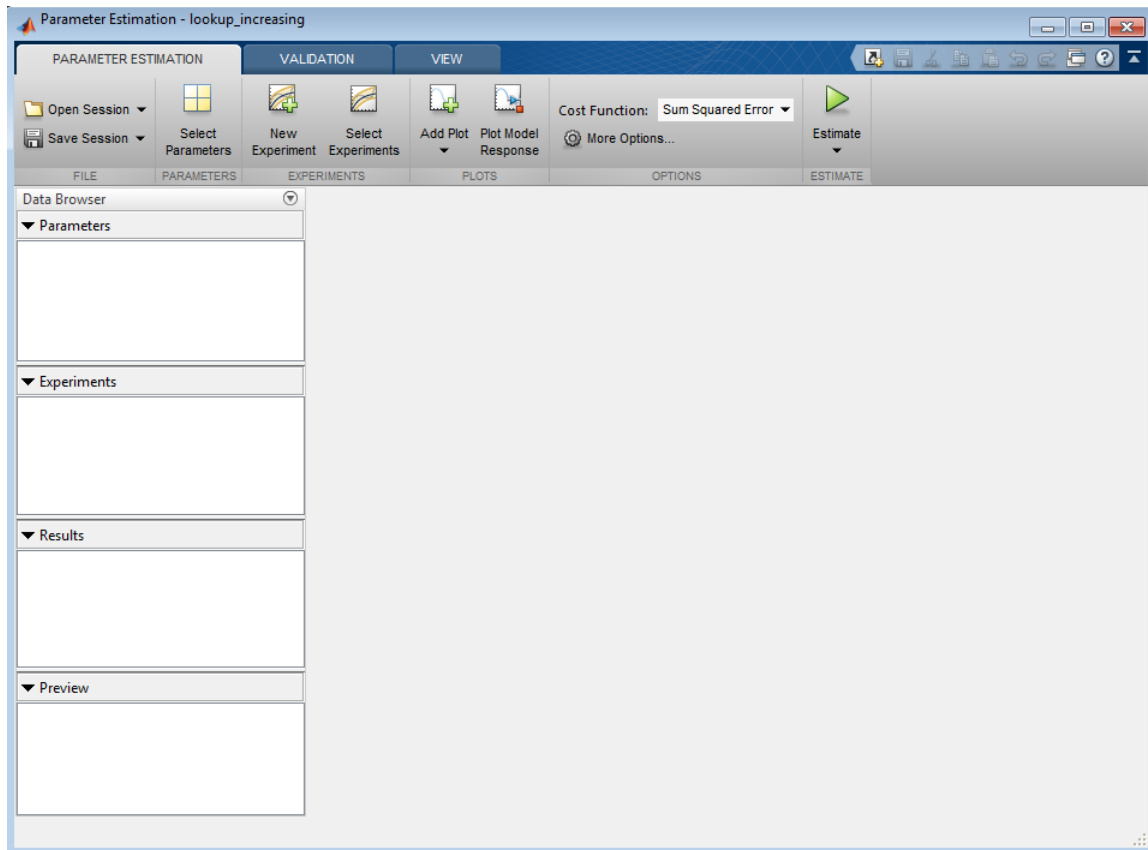
Estimate the Monotonically Increasing Table Values Using Default Settings

After you open a session for parameter estimation, as described in "Open a Parameter Estimation Session" on page 6-6, use the following steps to estimate the constrained lookup table values:

1   To create a new experiment, on the **Parameter Estimation** tab, click **New Experiment**. Name it EstimationData. Then import the I/O data, xdata1 and ydata1, and the time vector, time1 into the experiment. To do this, open the experiment editor by right-clicking EstimationData and selecting **Edit...**. Then, type [time1,ydata1] in the output dialog box and [time1,xdata1] in the input

dialog box. For more information, see "Import Data" on page 1-6. After you import the data, the experiment looks as follows:
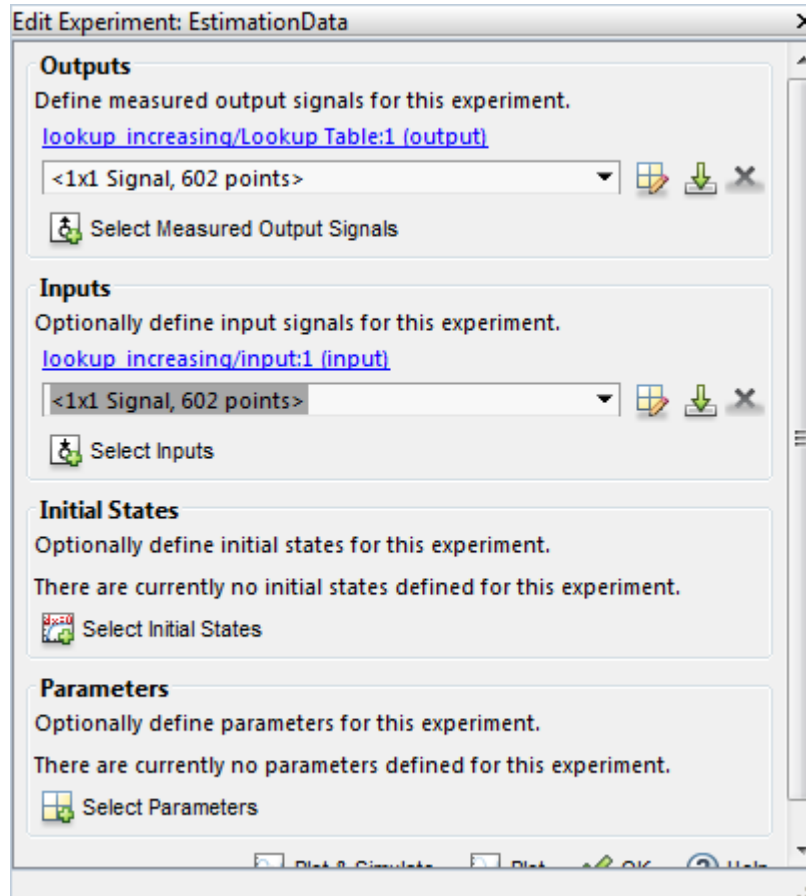
2. Run an initial simulation to view the measured data, simulated table values, and the initial table values by typing the following commands at the MATLAB prompt:

```
sim('lookup_increasing')
figure(1); plot(xdata1,ydata1, 'm*', xout, yout,'b^')
hold on; plot(-5:5, cumsum(table), 'k', 'LineWidth', 2)
```
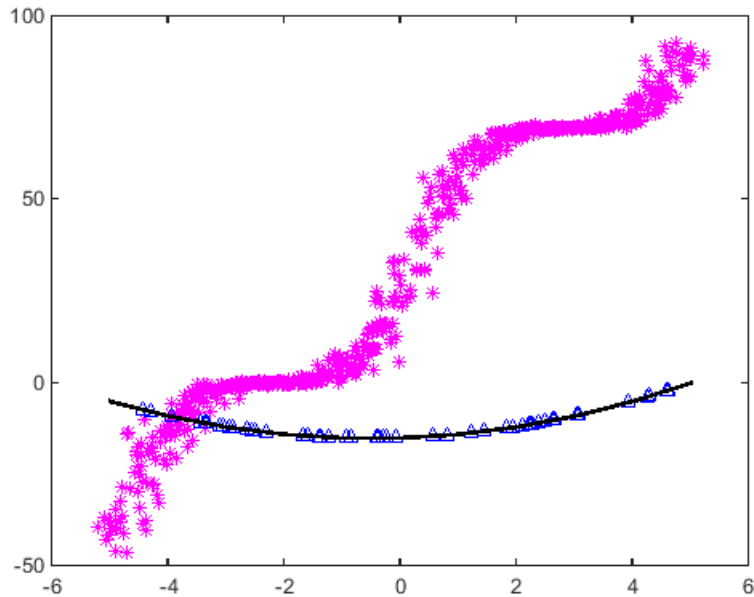
The x-axis and y-axis represent the input and output data, respectively. The figure shows the following plots:

- Measured data — Represented by the magenta stars (*).

  Note: As described in "About the Data" on page 6-6, the output data is a monotonically increasing function of the input data.

- Initial table values — Represented by the black line.
- Initial simulation data — Represented by the blue deltas (Δ).

You can see that the initial table values and simulated data do not match with the measured data.

3   To select the table values to estimate, on the **Parameter Estimation** tab, click **Select Parameters**. This opens the **Edit:Estimated Parameters** dialog. In the **Parameters Tuned for all Experiments** panel, click **Select parameters** to

launch the Select Model Variables dialog. Check the box next to `table`, and click
OK.
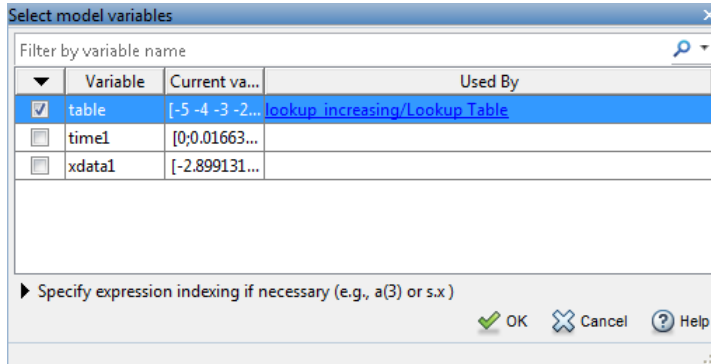
The **Edit:Estimated Parameters** window looks as follows. The table values are
selected for estimation by default.
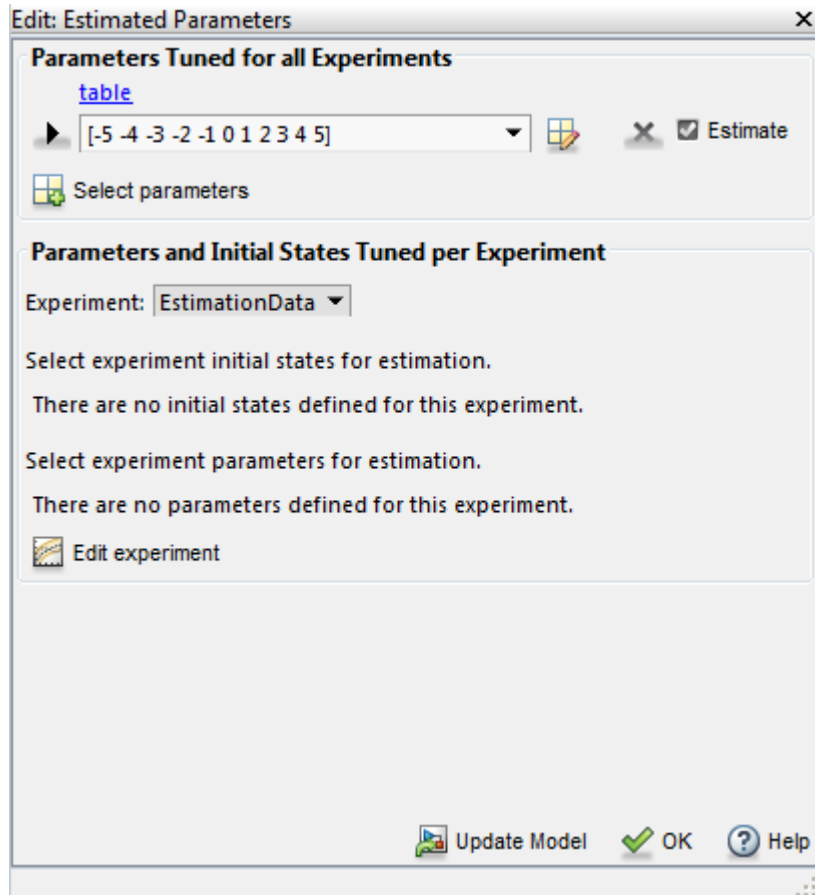
4. On the **Parameter Estimation** tab, click **Select Experiment**. By default, EstimationData is selected for estimation. If not, check the box under the **Estimation** column, and click **OK**.
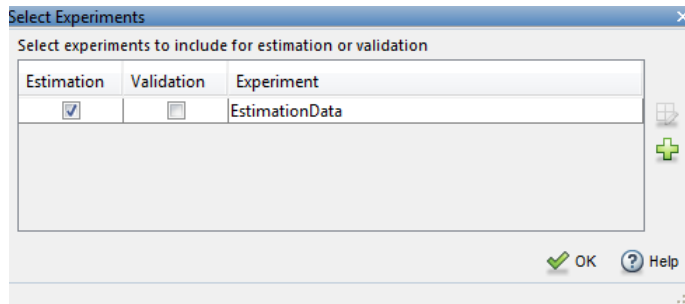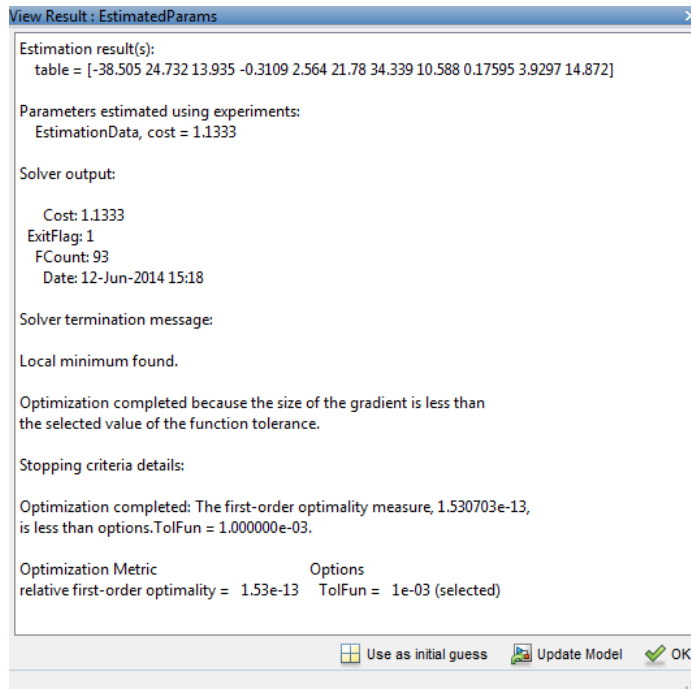
5 To estimate the table values using the default settings, on the **Parameter Estimation** tab, click **Estimate** to open the **Parameter Trajectory** plot and **Estimation Progress Report** window. The **Parameter Trajectory** plot shows the change in the parameter values at each iteration.

After the estimation converges, the **Parameter Trajectory** plot looks like this:
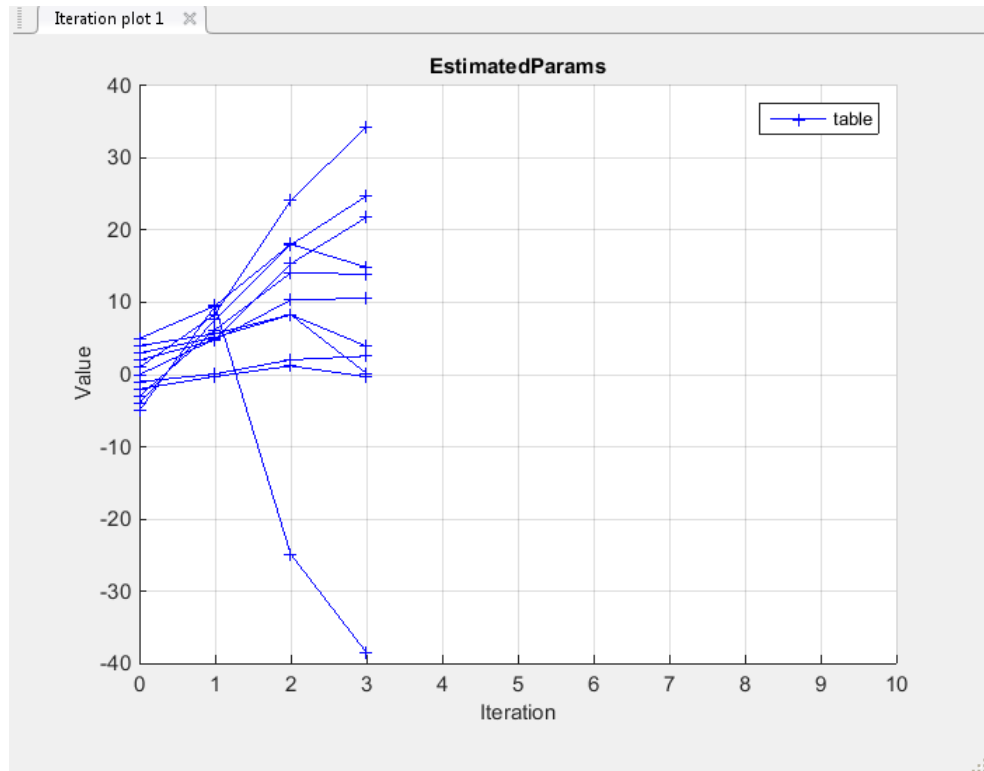
The **Estimation Progress Report** shows the iteration number, the number of times the objective function is evaluated, and the value of the cost function at the end of each iteration. After the estimation converges, the **Estimation Progress Report** looks like this:

The estimated parameters are saved in `EstimatedParams` in the **Results** section of the **Data Browser** pane on the left. To view the results right-click `EstimatedParams` and select **Open...**. The report looks like this:

View Result : EstimatedParams

```
Estimation result(s):
    table = [-38.505 24.732 13.935 -0.3109 2.564 21.78 34.339 10.588 0.17595 3.9297 14.872]

Parameters estimated using experiments:
    EstimationData, cost = 1.1333

Solver output:

    Cost: 1.1333
    ExitFlag: 1
    FCount: 93
    Date: 12-Jun-2014 15:18

Solver termination message:

Local minimum found.

Optimization completed because the size of the gradient is less than
the selected value of the function tolerance.

Stopping criteria details:

Optimization completed: The first-order optimality measure, 1.530703e-13,
is less than options.TolFun = 1.000000e-03.

Optimization Metric                Options
relative first-order optimality =  1.53e-13    TolFun =  1e-03 (selected)
```

Use as initial guess   Update Model   OK

Validate the Estimation Results

After you estimate the table values, as described in "Estimate the Monotonically Increasing Table Values Using Default Settings" on page 6-9, you use another data set to validate that you have not over-fitted the model. You can plot and examine the following plots to validate the estimation results:

- Residuals plot
- Measured and simulated data plots

To validate the estimation results:

1   Create a new experiment to use for validation. Name it `ValidationData`.

2   Import the validation I/O data, `xdata2` and `ydata2`, and time vector, `time2` in the `ValidationData` experiment. To do this, open the experiment editor by right-clicking `ValidationData` and selecting **Edit...**. Then, type `[time2,ydata2]`

in the output dialog box and [time2,xdata2] in the input dialog box. For more information, see "Import Data" on page 1-6.

3 To select the experiment for validation, click **Select Experiments** on the **Parameter Estimation** tab. The ValidationData experiment is selected for estimation by default. Deselect the box for estimation and check for validation.
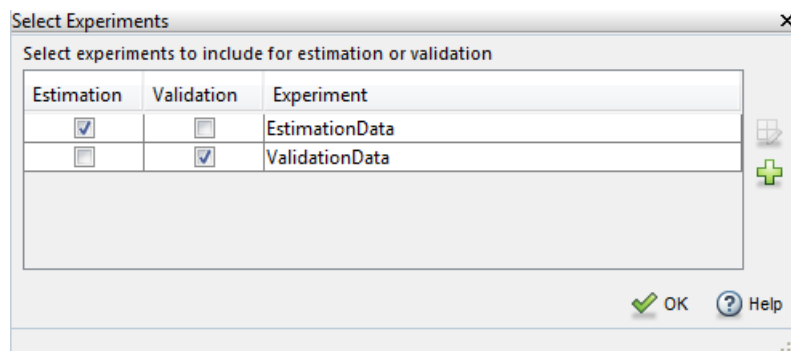
4 To select results to use, on the **Validation** tab, click **Select Results**. Deselect Use current parameter values and select EstimatedParams, and then click **OK**.
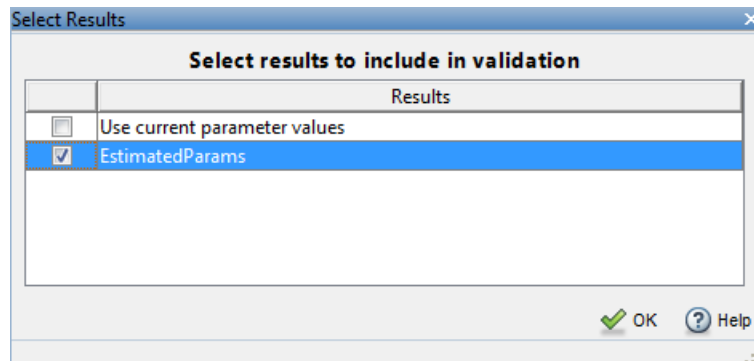
5 The Parameter Estimation tool, displays the experiment plot after validation by default. Add the residuals plot by checking the corresponding box on the **Validation** tab.

To start validation, on the **Validation** tab, click **Validate**.

You can see that the data simulated using the estimated parameters agrees with the measured data.

6   Examine the plots.

a   The experiment plot shows the data simulated using the estimated parameters with the measured validation data.
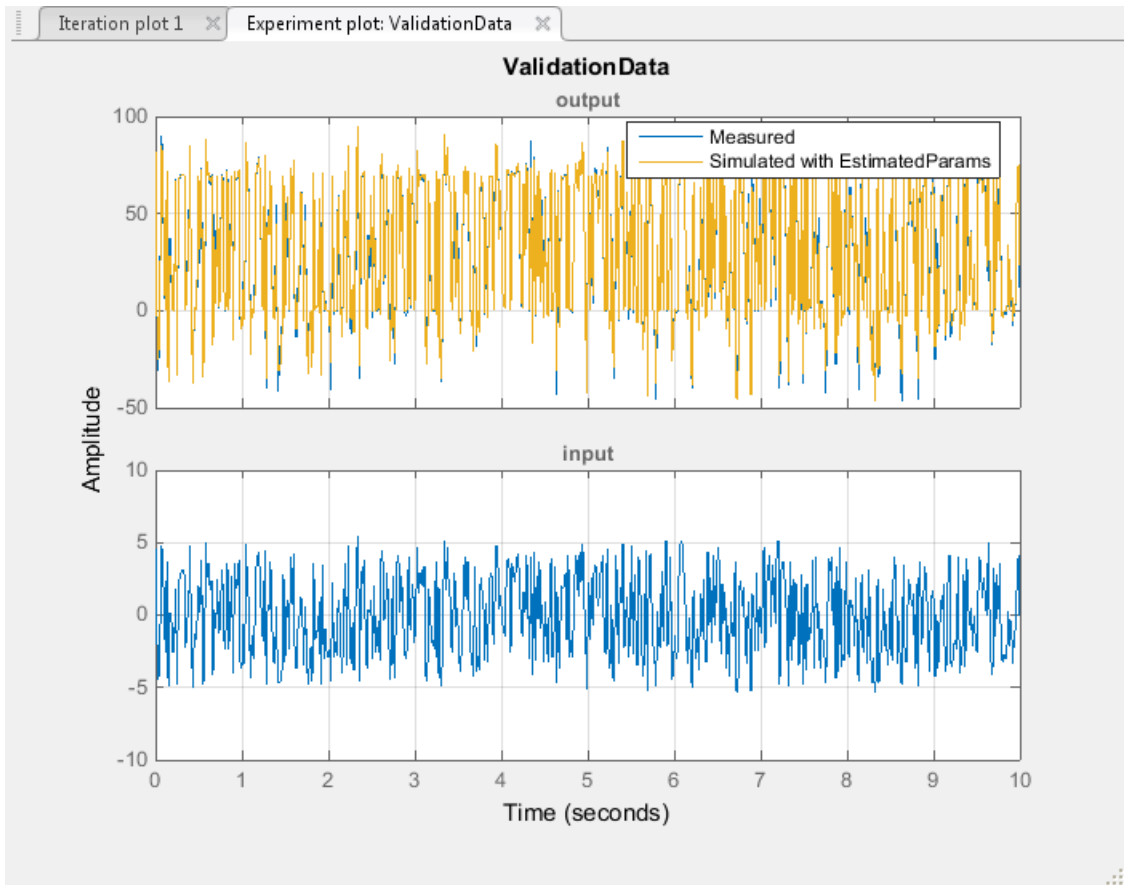
Both data agree closely with each other.

b   Click Residual plot: ValidationData to open the residuals plot.
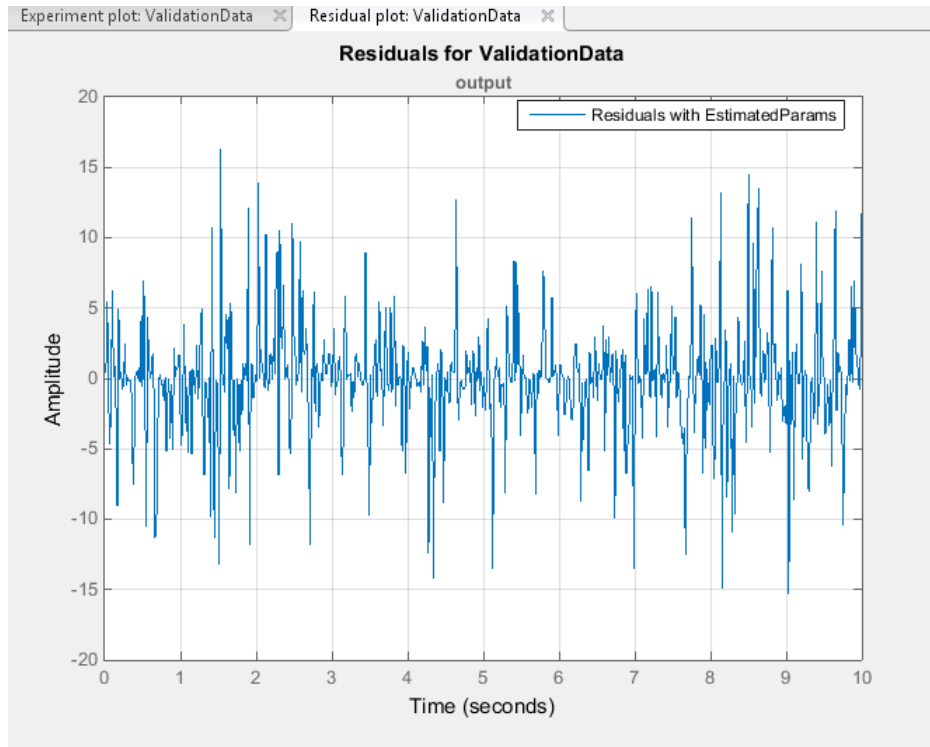
The residuals, which show the difference between the simulated and measured data, lie the range [-15,15]— within 20% of the maximum output variation. This indicates a good match between the measured and the simulated table data values.

c   Plot and examine the validation data, simulated data, and estimated table values.

```
sim('lookup_increasing')
figure(2); plot(xdata2,ydata2, 'm*', xout, yout,'b^')
hold on; plot(-5:5, cumsum(table), 'k', 'LineWidth', 2)
```
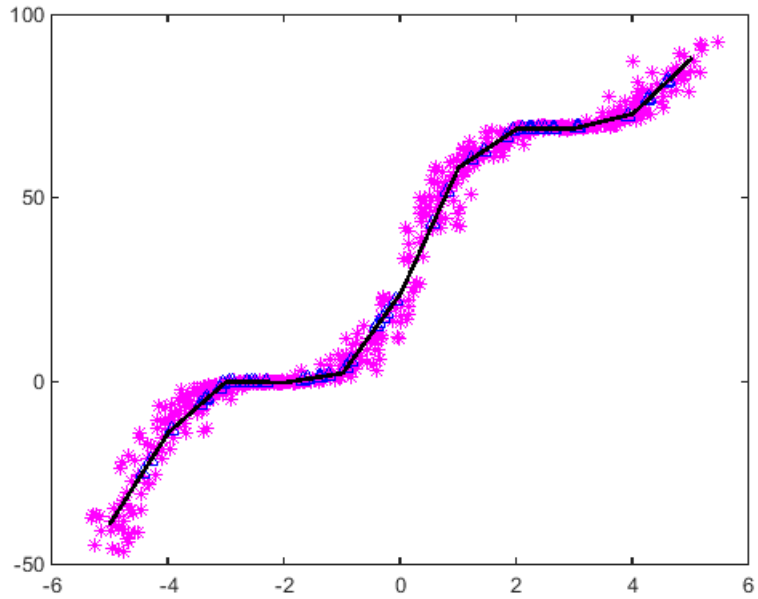
The plot shows that the table values, shown as the black line, match both the measured data and the simulated table values. The table data values cover the entire range of input values, which indicates that all the lookup table values have been estimated.

Estimate Lookup Table Values from Data

Objectives

This example shows how to estimate lookup table values from time-domain input-output (I/O) data.

About the Data

In this example, use the I/O data in `lookup_regular.mat` to estimate the values of a lookup table. The MAT-file includes the following variables:

- `xdata1` — Consists of 63 uniformly-sampled input data points in the range [0,6.5]
- `ydata1` — Consists of output data corresponding to the input data samples
- `time1` — Time vector

Use the I/O data to estimate the lookup table values in the `lookup_regular` Simulink model. The lookup table in the model contains ten values, which are stored in the MATLAB variable `table`. The initial table values comprise a vector of 0s. To learn more about how to model a system using lookup tables, see "Guidelines for Choosing a Lookup Table" in the Simulink documentation.

Open a Parameter Estimation Session

To estimate the lookup table values, open a Parameter Estimation session.

1   Open the lookup table model by typing the following command at the MATLAB prompt:

```
lookup_regular
```

This command opens the Simulink model, and loads the estimation data into the MATLAB workspace.
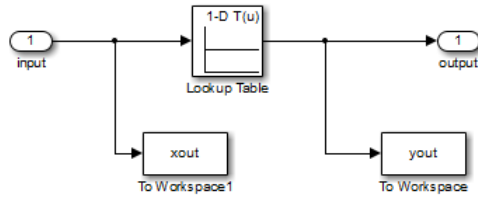
2 In the Simulink model, select **Analysis** > **Parameter Estimation** to open a new session with name **lookup_regular** in the Parameter Estimation tool.
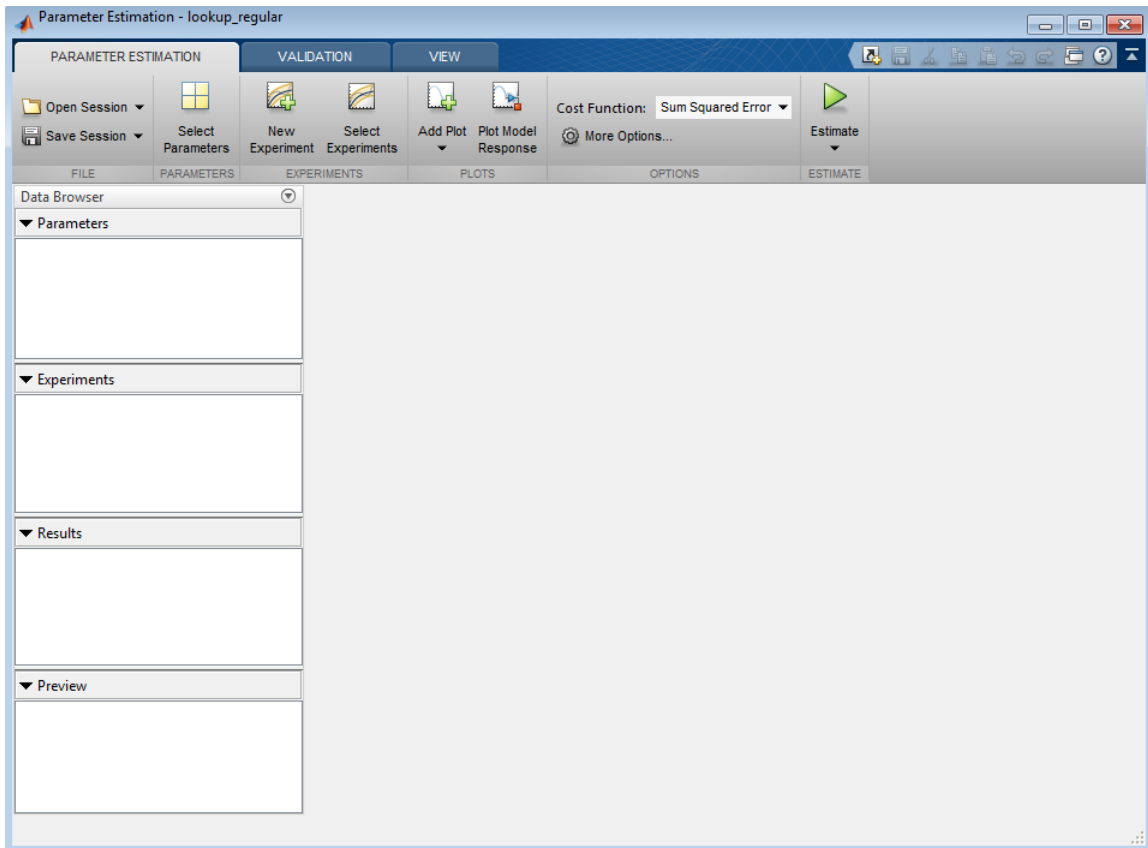
Estimate the Table Values Using Default Settings

Use the following steps to estimate the lookup table values.

1 Create a new experiment by clicking **New Experiment** on the **Parameter Estimation** tab. Name it `EstimationData`. Then import the I/O data, `xdata1` and `ydata1`, and the time vector, `time1`, into the experiment. To do this open the experiment editor by right-clicking `EstimationData` and selecting **Edit...**. Type `[time1,ydata1]` in the output dialog box and `[time1,xdata1]` in the input dialog box in the experiment editor. For more information, see "Import Data" on page 1-6. After you import the data the experiment looks as follows:
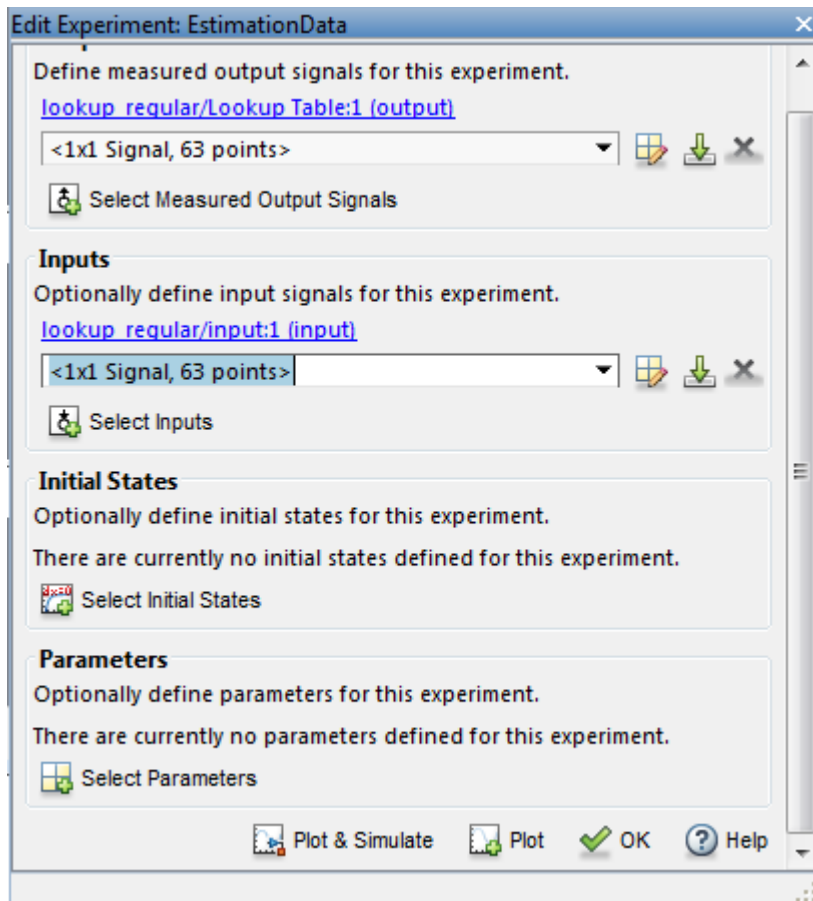
2 Run an initial simulation to view the I/O data, simulated output, and the initial table values. To do so, type the following commands at the MATLAB prompt:

```
sim('lookup_regular')
figure(1); plot(xdata1,ydata1, 'm*', xout, yout,'b^')
hold on; plot(linspace(0,6.5,10), table, 'k', 'LineWidth', 2);
legend('Measured data','Initial simulation data','Initial table values');
```
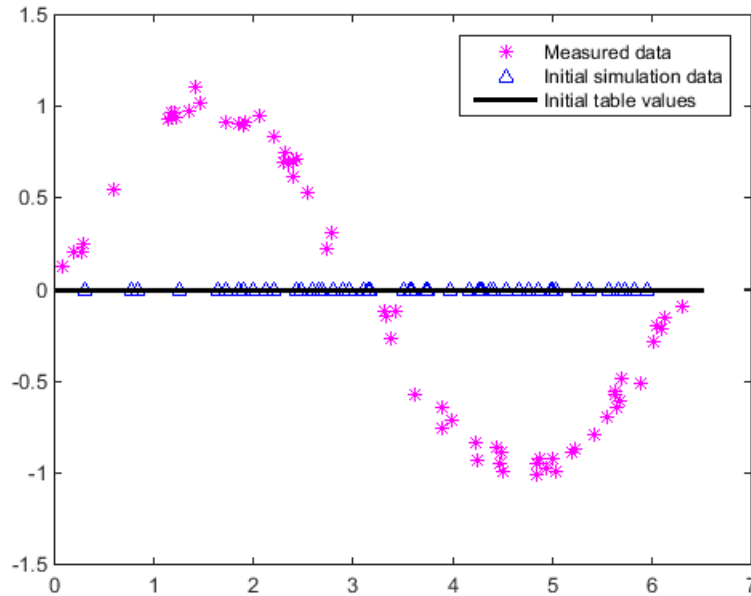
The x-axis and y-axis of the figure represent the input and output data, respectively. The figure shows the following plots:

- Measured data — Represented by the magenta stars (*).

  Note: As described in "About the Data" on page 6-6, the output data is a monotonically increasing function of the input data.

- Initial table values — Represented by the black line.
- Initial simulation data — Represented by the blue deltas (Δ).

You can see that the initial table values and simulated data do not match with the measured data.

3 To select the table values to estimate, on the **Parameter Estimation** tab, click the **Select Parameters** button to open the **Edit:Estimated Parameters** dialog. In

the **Parameters Tuned for all Experiments** panel, click **Select parameters** to launch the Select Model Variables dialog. Check the box next to table, and click **OK**.
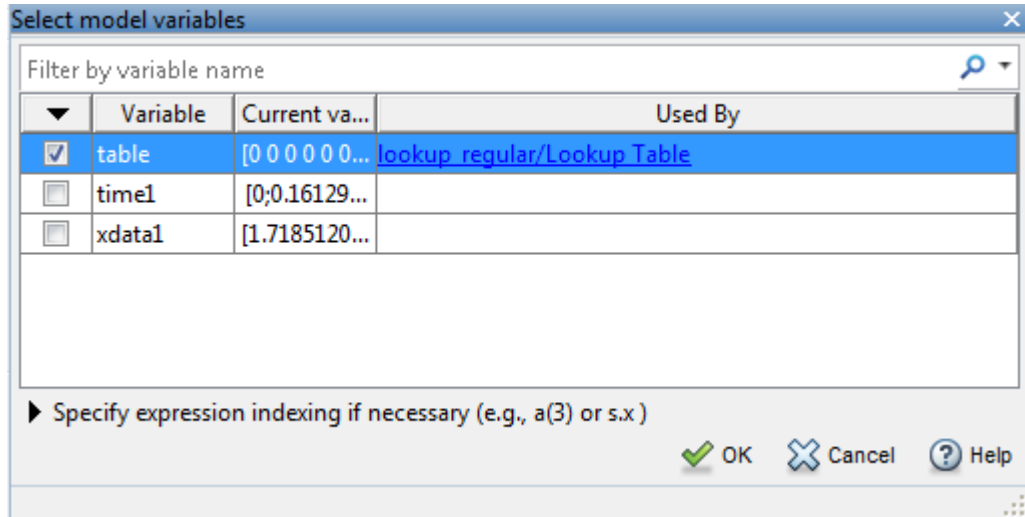
The **Edit:Estimated Parameters** window now looks as follows. The table values are selected for estimation by default.
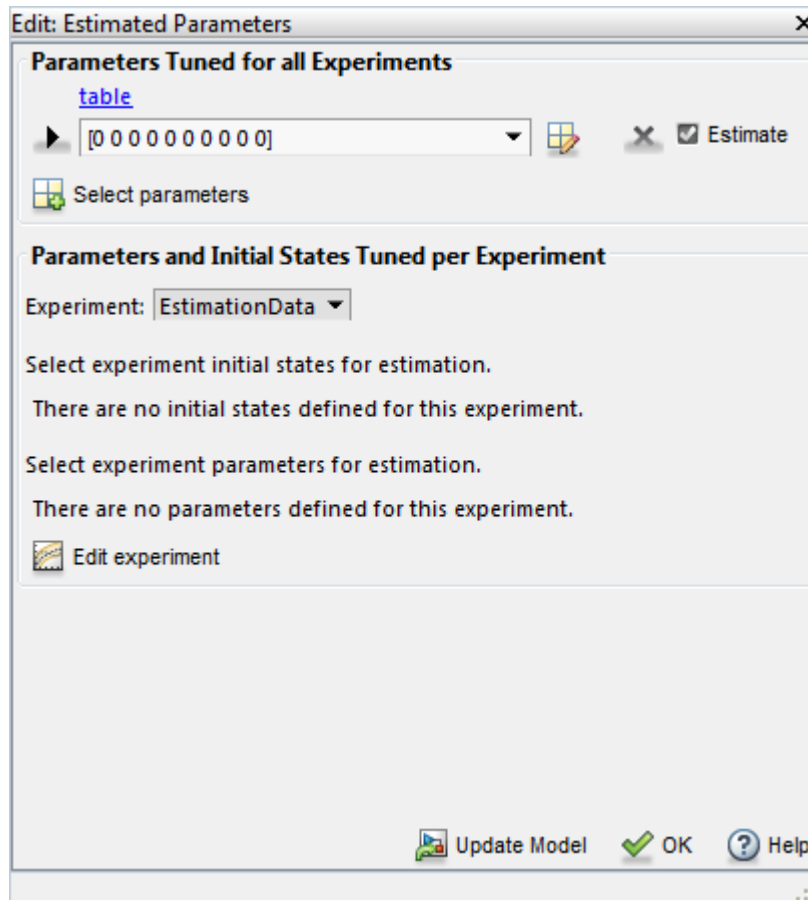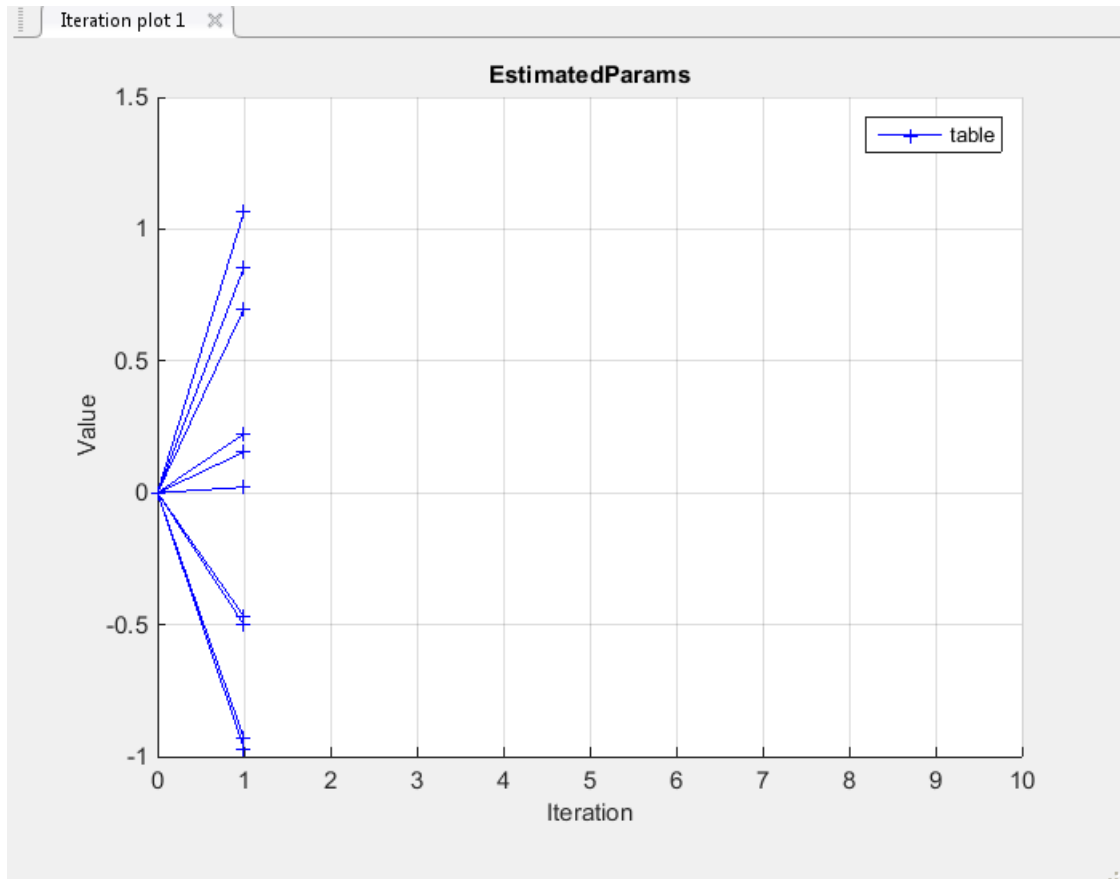
4   On the **Parameter Estimation** tab, click **Select Experiment**. `EstimationData` is selected for estimation by default. If not, check the box under the **Estimation** column, and click **OK**.

5   To estimate the table values using the default settings, on the **Parameter Estimation** tab, click **Estimate** to open the **Parameter Trajectory** plot and **Estimation Progress Report** window. The **Parameter Trajectory** plot shows the change in the parameter values at each iteration.

After the estimation converges, the **Parameter Trajectory** plot looks like this:

The **Estimation Progress Report** shows the iteration number, number of times the objective function is evaluated, and the value of the cost function at the end of each iteration. After the estimation converges, the **Estimation Progress Report** looks like this:

The estimated parameters are saved in `EstimatedParams` in the **Results** section of the **Data Browser** pane on the left. To view the results, right-click on `EstimatedParams` and then select **Open...**. The report resembles the following.
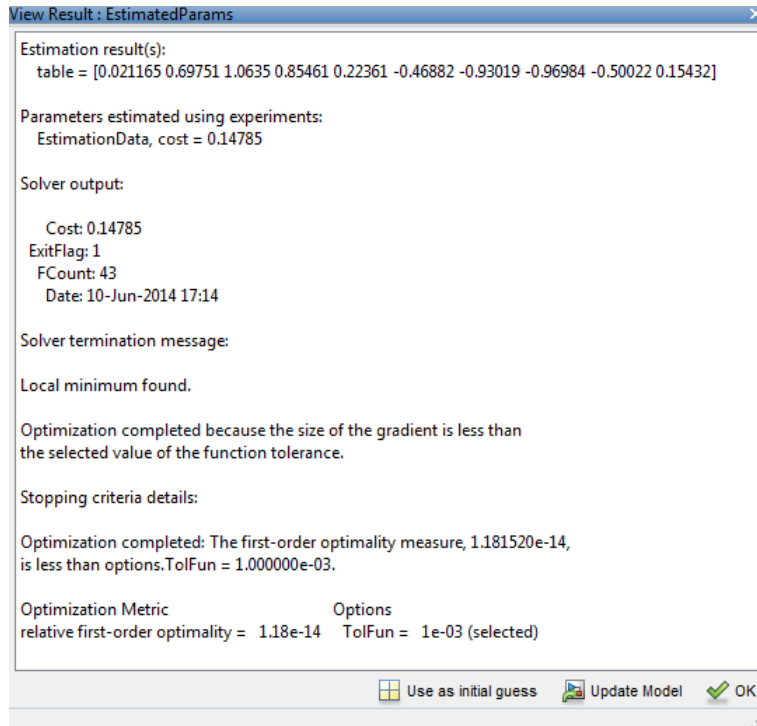
This report includes the estimated parameter values, the final value of the cost function, and other optimization results. You can see that the optimization stopped when the size of the gradient, 1.18e-14 was less than the criteria value, 1e-3.
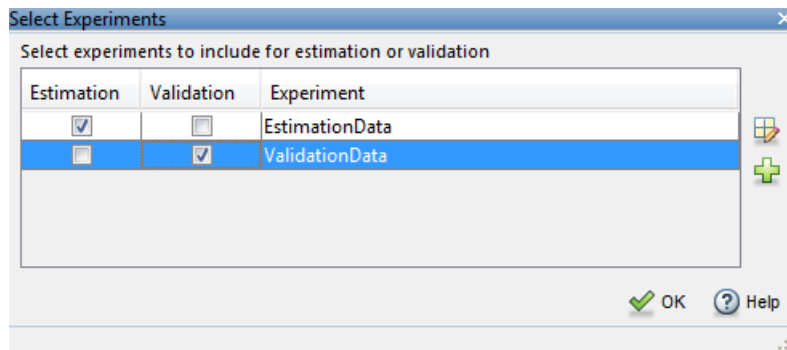
Validate the Estimation Results

After you estimate the table values, as described in "Estimate the Table Values Using Default Settings" on page 6-25, you must use another data set to validate that you have not over-fitted the model. You can plot and examine the following plots to validate the estimation results:
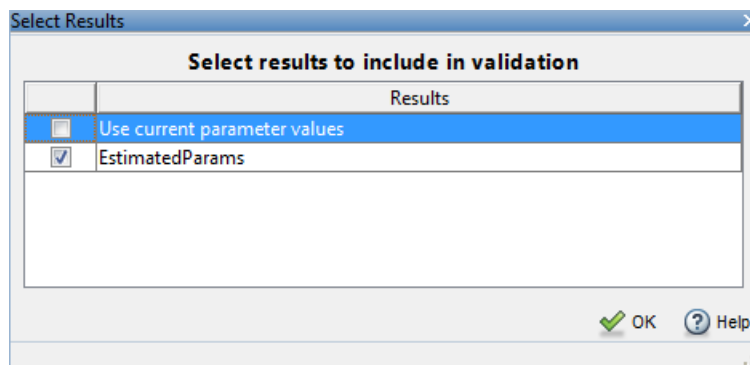
- Residuals plot
- Measured and simulated data plots

To validate the estimation results:

1 Create a new experiment to use for validation. Name it `ValidationData`. Import the validation I/O data, `xdata2` and `ydata2`, and time vector, `time2` in the `ValidationData` experiment. To do this open the experiment editor by right-clicking `ValidationData` and selecting **Edit...**. Then, type `[time2,ydata2]` in the output dialog box and `[time2,xdata2]` in the input dialog box in the experiment editor. For more information, see "Import Data" on page 1-6.

2 To select the experiment for validation, on the **Parameter Estimation** tab, click **Select Experiments**. The `ValidationData` experiment is selected for estimation by default. Deselect the box for estimation and check it for validation.
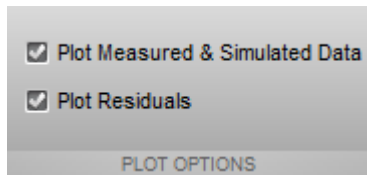

3 To select results to use, on the **Validation** tab, click **Select Results**. Deselect `Use current parameter values` and select `EstimatedParams`, and click **OK**.
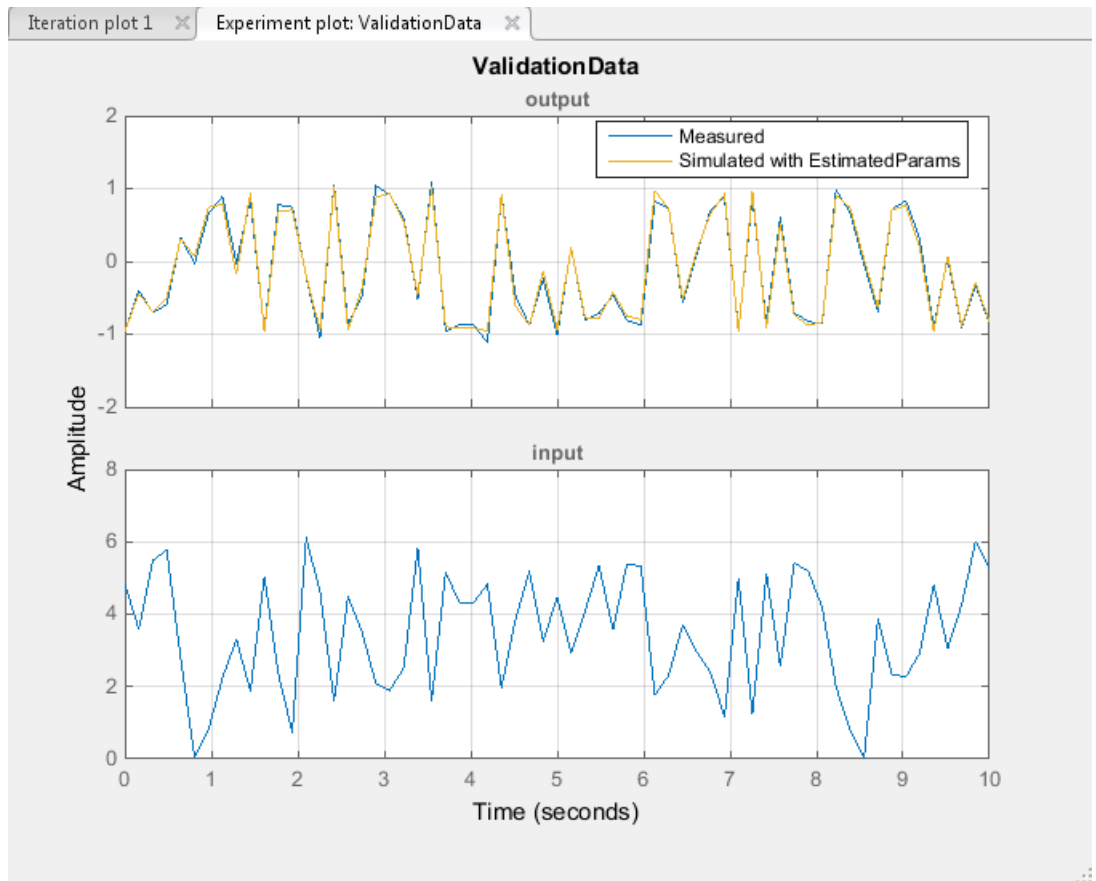

4 The Parameter Estimation tool, by default, displays the experiment plot after validation. Add the residuals plot by checking the corresponding box on the **Validation** tab.
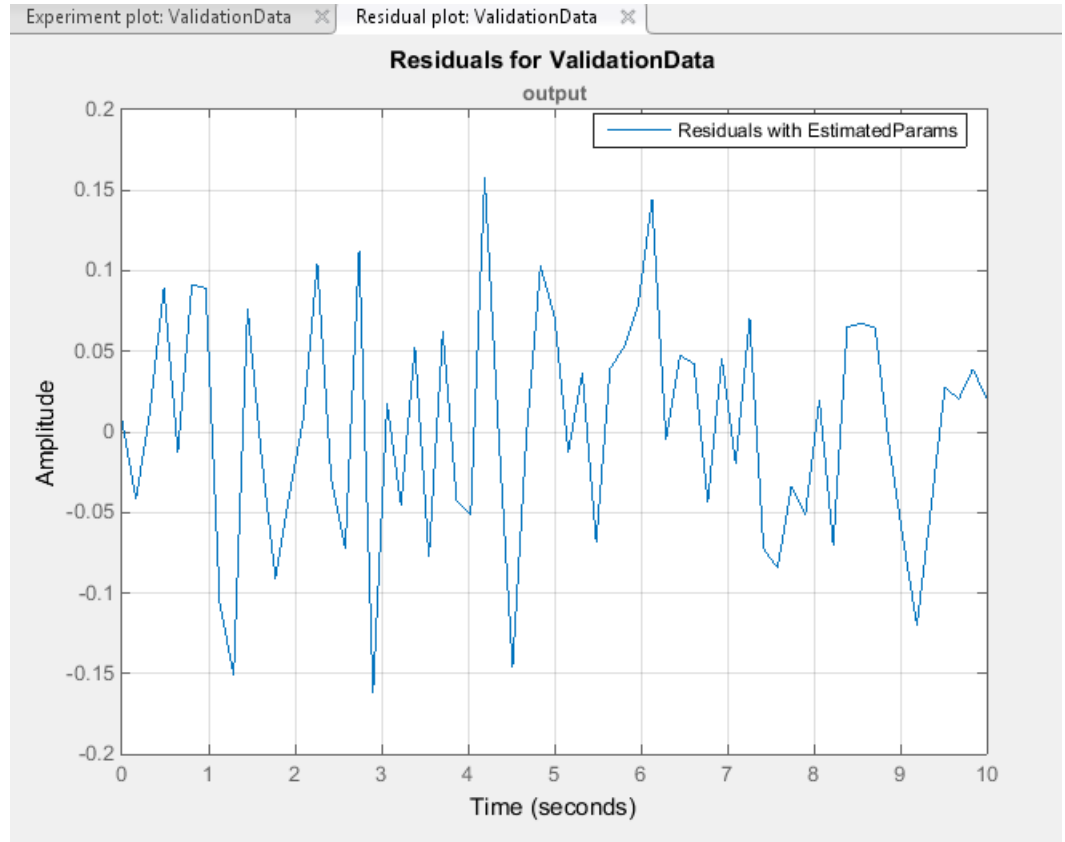
To start validation, on the **Validation** tab, click **Validate**.

5   Examine the plots

 a Experiment plot

You can see that the data simulated using the estimated parameters agrees with the measured validation data.

b Click Residual plot: ValidationData to open the residuals plot.

The residuals, which show the difference between the simulated and measured data, lie in the range [-0.15,0.15]— within 15% of the maximum output variation. This indicates a good match between the measured and the simulated table data values.

c Plot and examine the estimated table values against the validation data set and the simulated table values by typing the following commands at the MATLAB prompt.

```
sim('lookup_regular')
figure(2); plot(xdata2,ydata2, 'm*', xout, yout,'b^')
hold on; plot(linspace(0,6.5,10), table, 'k', 'LineWidth', 2)
```
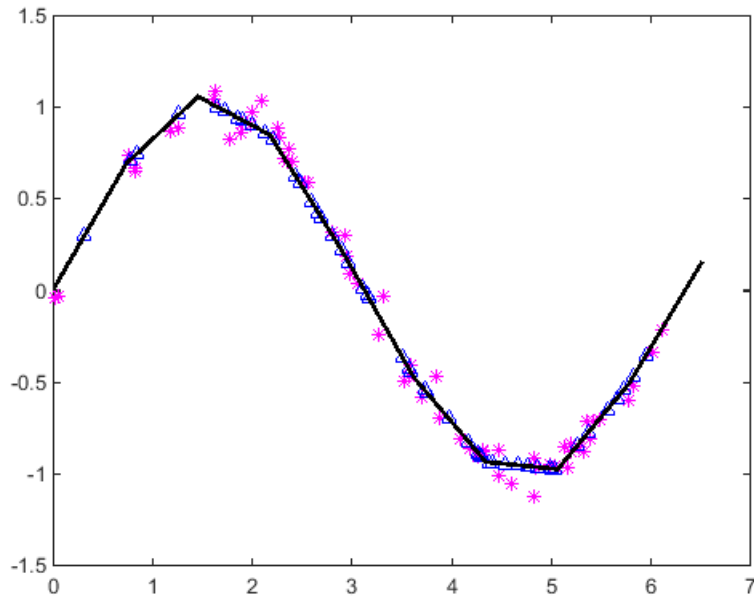

The plot shows that the table values, displayed as the black line, match both the validation data and the simulated table values. The table data values cover the entire range of input values, which indicates that all the lookup table values have been estimated.

Building Models Using Adaptive Lookup Table Blocks

Simulink Design Optimization software provides blocks for modeling systems as adaptive lookup tables. You can use the adaptive lookup table blocks to create lookup tables from measured or simulated data. You build a model using the adaptive lookup table blocks, and then simulate the model to adapt the lookup table values to the time-varying I/O data. During simulation, the software uses the input data to locate the table values, and then uses the output data to recalculate the table values. The updated table values are stored in the adaptive lookup table block. For more information, see "What are Adaptive Lookup Tables?" on page 6-2.

The Adaptive Lookup Table library has the following blocks:

- Adaptive Lookup Table (1D Stair-Fit) — One-dimensional adaptive lookup table
- Adaptive Lookup Table (2D Stair-Fit) — Two-dimensional adaptive lookup table
- Adaptive Lookup Table (nD Stair-Fit) — Multidimensional adaptive lookup table

---

Note: Use the Adaptive Lookup Table (nD Stair-Fit) block to create lookup tables of three or more dimensions.

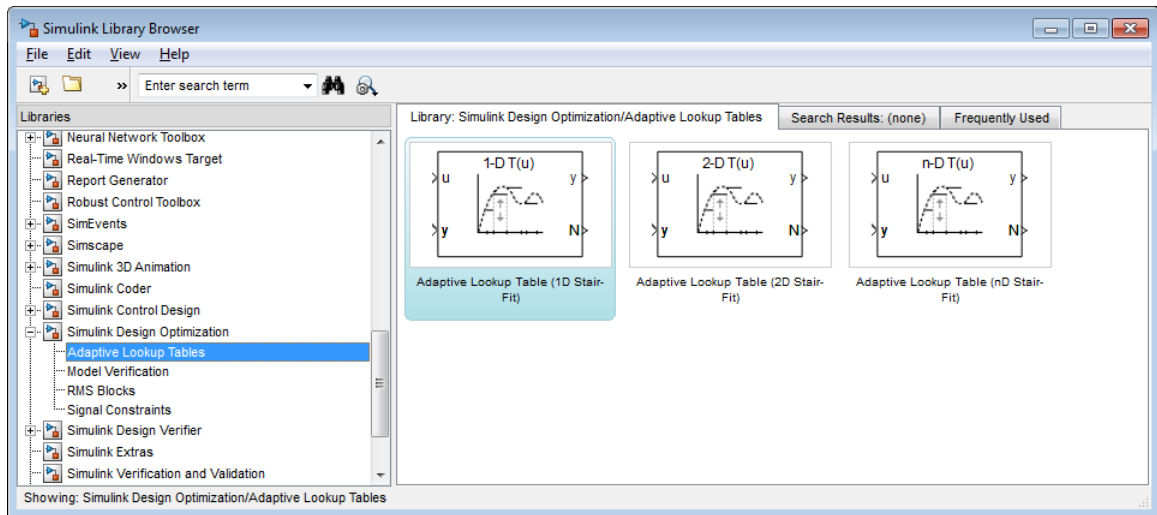To access the Adaptive Lookup Tables library:

1 Open the Simulink Library Browser.

At the MATLAB prompt, enter `simulink`.

2 Open the Simulink Design Optimization library.

In the **Libraries** pane, expand the **Simulink Design Optimization** node.

3 In the Simulink Design Optimization library tree, click **Adaptive Lookup Tables**.

By default, the Adaptive Lookup Table blocks have two inputs and outputs. You can display additional inputs and outputs in a block by selecting the corresponding options in the Function Block Parameters dialog box. To learn more about the options, see the block reference pages.
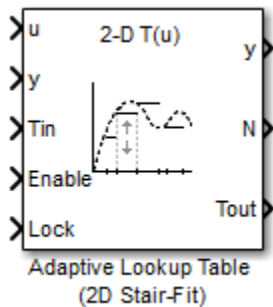
Adaptive Lookup Table Block Showing Inputs and Outputs

The 2-D Adaptive Lookup Table block has the following inputs and outputs:

- u and y — Input and output data of the system being modeled, respectively.

For example, to model an engine's efficiency as a function of engine rpm and manifold pressure, specify u as the rpm, y as the pressure, and y   as the efficiency signals.

- Tin — The initial table data.
- Enable — Signal to enable, disable, or reset the adaptation process.
- Lock — Signal to update only specified cells in the table.
- y — Value of the cell currently being adapted.
- N — Number of the cell currently being adapted.
- Tout — Values of the adapted table data.

For more information on how to use adaptive lookup tables, see "Model Engine Using n-D Adaptive Lookup Table" on page 6-44.

A typical Simulink diagram using an adaptive lookup table block is shown in the next figure.
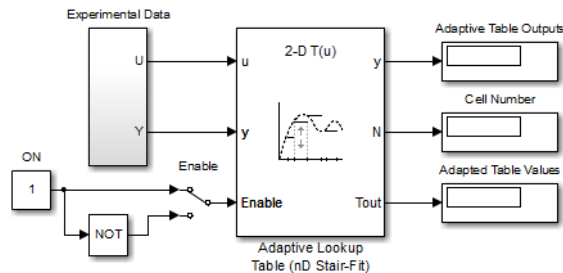
Simulink Diagram Using an Adaptive Lookup Table

In this figure, the Experiment Data block imports a set of experimental data into Simulink through MATLAB workspace variables. The initial table is specified in the block mask parameters. When the simulation runs, the initial table begins to adapt to new data inputs and the resulting table is copied to the block's output.

See Also

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

Related Examples

- "Model Engine Using n-D Adaptive Lookup Table" on page 6-44

More About

Selecting an Adaptation Method

You specify the algorithm using the **Adaptation Method** drop-down list in the Function Block Parameters dialog box of an adaptive lookup table block. This section discusses the details of these algorithms.

Sample Mean

`Sample mean` provides the average value of n output data samples and is defined as:

$$\hat{y}(n) = \frac{1}{n} \sum_{i=1}^{n} y(i)$$

where $y(i)$ is the ith measurement collected within a particular *cell*. For each input data u, the sample mean at the corresponding cell is updated using the output data measurement, y. Instead of accumulating n samples of data for each cell, a recursive relation is used to calculate the sample mean. The recursive expression is obtained by the following equation:

$$\hat{y}(n) = \frac{1}{n} \left[\sum_{i=1}^{n-1} y(i) + y(n) \right] = \frac{n-1}{n} \left[\frac{1}{n-1} \sum_{i=1}^{n-1} y(i) \right] + \frac{1}{n} y(n) = \frac{n-1}{n} \hat{y}(n-1) + \frac{1}{n} y(n)$$

where $y(n)$ is the nth data sample.

Defining *a priori estimation error* as $e(n) = y(n) - \hat{y}(n-1)$, the recursive relation can be written as:

$$\hat{y}(n) = \hat{y}(n-1) + \frac{1}{n} e(n)$$

where $n \geq 1$ and the initial estimate $\hat{y}(0)$ is arbitrary.

In this expression, only the number of samples, n, for each cell— rather than n data samples—is stored in memory.

Sample Mean with Forgetting

The adaptation method "Sample Mean" on page 6-42 has an *infinite memory*. The past data samples have the same weight as the final sample in calculating the sample mean. `Sample mean (with forgetting)` uses an algorithm with a *forgetting factor* or **Adaptation gain** that puts more weight on the more recent samples. This algorithm provides robustness against initial response transients of the plant and an adjustable speed of adaptation. `Sample mean (with forgetting)` is defined as:

$$\hat{y}(n) = \frac{1}{\sum_{i=1}^{n} \lambda^{n-i}} \sum_{i=1}^{n} \lambda^{n-i} y(i)$$

$$= \frac{1}{\sum_{i=1}^{n} \lambda^{n-i}} \left[\sum_{i=1}^{n-1} \lambda^{n-i} y(i) + y(n) \right] = \frac{s(n-1)}{s(n)} \hat{y}(n-1) + \frac{1}{s(n)} y(n)$$

where $\lambda \in [0,1]$ is the **Adaptation gain** and $s(k) = \sum_{i=1}^{k} \lambda^{n-i}$.

Defining *a priori estimation error* as $e(n) = y(n) - \hat{y}(n-1)$, where $n \geq 1$ and the initial estimate $\hat{y}(0)$ is arbitrary, the recursive relation can be written as:

$$\hat{y}(n) = \hat{y}(n-1) + \frac{1}{s(n)} e(n) = \hat{y}(n-1) + \frac{1-\lambda}{1-\lambda^n} e(n)$$

A small value of λ results in faster adaptation. A value of 0 indicates short memory (last data becomes the table value), and a value of 1 indicates long memory (average all data received in a cell).

Model Engine Using n-D Adaptive Lookup Table

In this section...
"Objectives" on page 6-44
"About the Data" on page 6-44
"Building a Model Using Adaptive Lookup Table Blocks" on page 6-45
"Adapting the Lookup Table Values Using Time-Varying I/O Data" on page 6-54

Objectives

In this example, you learn how to capture the time-varying behavior of an engine using an n-D adaptive lookup table. You accomplish the following tasks using the Simulink software:

- Configure an adaptive lookup table block to model your system.
- Simulate the model to update the lookup table values dynamically.
- Export the adapted lookup table values to the MATLAB workspace.
- Lock a specific cell in the table during adaptation.
- Disable the adaptation process and use the adaptive lookup table as a static lookup table.

About the Data

In this example, you use the data in `vedata.mat` which contains the following variables measured from an engine:

- `X` — 10 input breakpoints for intake manifold pressure in the range [10,100]
- `Y` — 36 input breakpoints for engine speed in the range [0,7000]
- `Z` — 10x36 matrix of table data for engine volumetric efficiency

  To learn more about breakpoints and table data, see "Anatomy of a Lookup Table" in the Simulink documentation.

The output volumetric efficiency of the engine is time varying, and a function of two inputs—intake manifold pressure and engine speed. The data in the MAT-file is used to generate the time-varying input and output (I/O) data for the engine.

Building a Model Using Adaptive Lookup Table Blocks

In this portion of the tutorial, you learn how to build a model of an engine using an Adaptive Lookup Table block.

1   Open a preconfigured Simulink model by typing the model name at the MATLAB prompt:

```
enginetable1_data
```

The Experimental Data subsystem in the Simulink model generates time-varying I/O data during simulation.
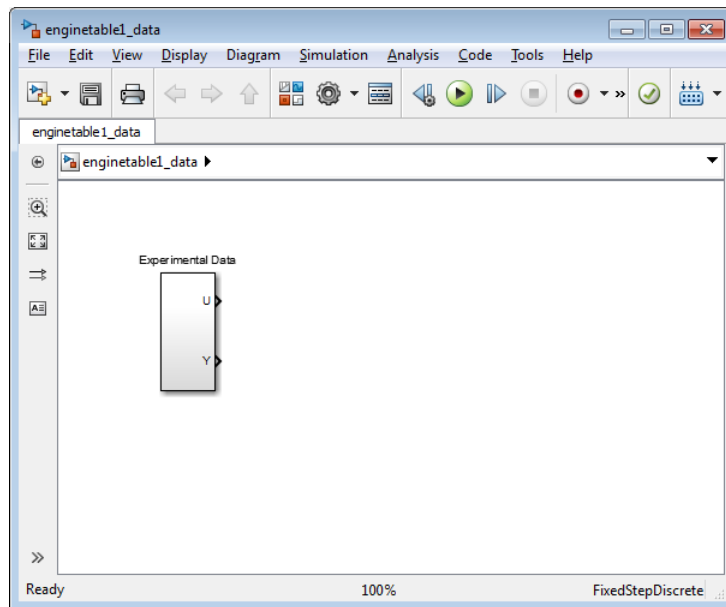
This command also loads the variables X, Y and Z into the MATLAB workspace. To learn more about this data, see "About the Data" on page 6-44.
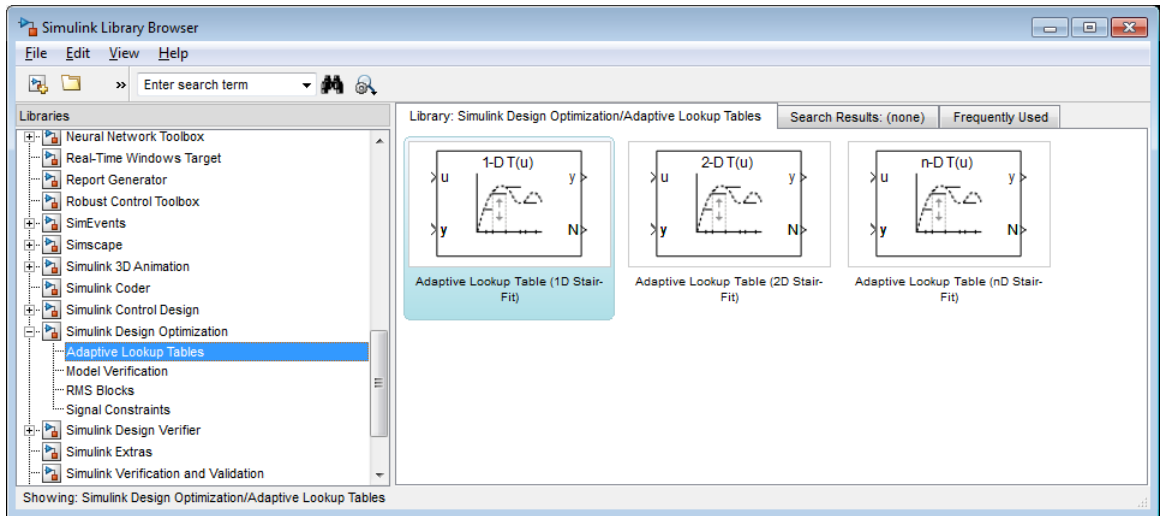
2   Add an Adaptive Lookup Table block to the Simulink model.

a   Open the Simulink Library Browser.

At the MATLAB prompt, enter simulink.

    b   Open the Simulink Design Optimization library.

   In the **Libraries** pane, expand the **Simulink Design Optimization** node.

    c   In the Simulink Design Optimization library tree, click **Adaptive Lookup Tables**.

    d   Drag and drop the Adaptive Lookup Table (nD Stair-Fit) block from the Adaptive Lookup Tables library to the Simulink model window.
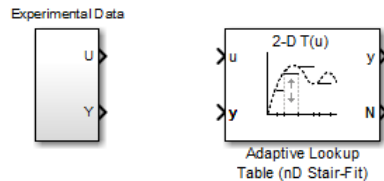
3   Double-click the Adaptive Lookup Table (nD Stair-Fit) block to open the Function Block Parameters: Adaptive Lookup Table (nD Stair-Fit) dialog box.
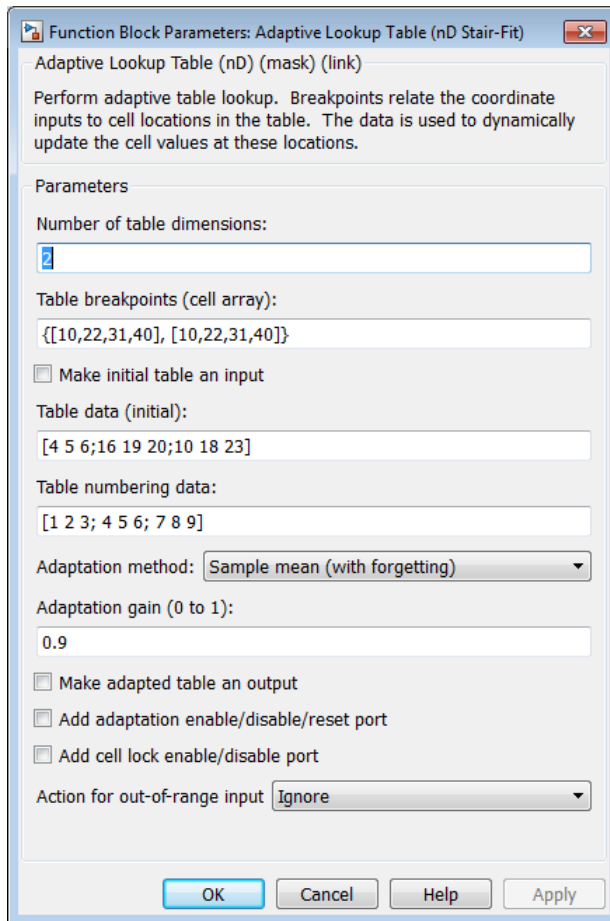
4 In the Function Block Parameters dialog box:

    a Specify the following block parameters:

- **Table breakpoints (cell array)** — Enter `{[X; 110], [Y; 7200]}` to specify the range of input breakpoints.

- **Table data (initial)** — Enter `rand(10,36)` to specify random numbers as the initial table values for the volumetric efficiency.

- **Table numbering data** — Enter `reshape(1:360,10,36)` to specify a numbering scheme for the table cells.

b Verify that `Sample mean (with forgetting)` is selected in the **Adaptation method** drop-down list.

c Enter `0.98` in the **Adaptation gain (0 to 1)** field to specify the *forgetting factor* for the `Sample mean (with forgetting)` adaptation algorithm.

An adaptation gain close to 1 indicates high robustness of the lookup table values to input noise. To learn more about the adaptation gain, see "Sample Mean with Forgetting" in "Selecting an Adaptation Method".

d Select the **Make adapted table an output** check box.

This action adds a new port named `Tout` to the Adaptive Lookup Table block. You use this port to plot the table values as they are being adapted.

e Select the **Add adaptation enable/disable/reset port** check box.

This action adds a new port named `Enable` to the Adaptive Lookup Table block. You use this port to enable or disable the adaptation process.

f Select the **Add cell lock enable/disable port** check box.

This action adds a new port named `Lock` to the Adaptive Lookup Table block. You use this port to lock a cell during the adaptation process.

g Verify that `Ignore` is selected in the **Action for out-of-range** drop-down list.
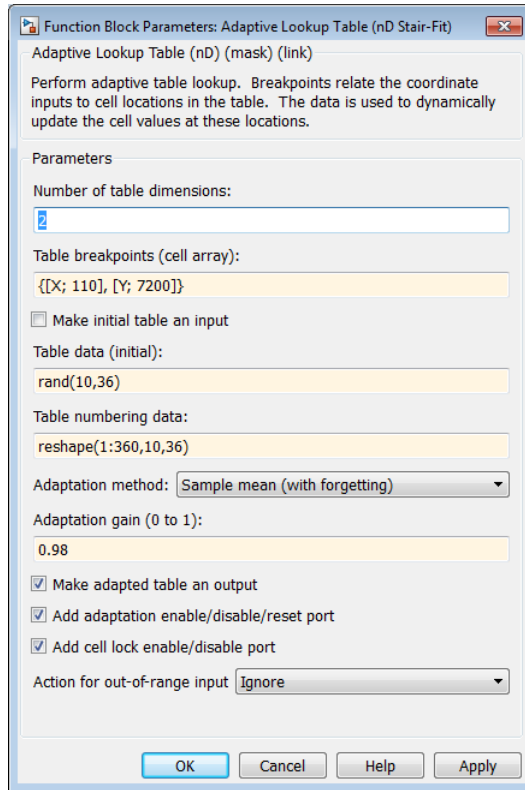
This selection specifies that the software ignores any time-varying inputs outside the range of input breakpoints during adaptation.

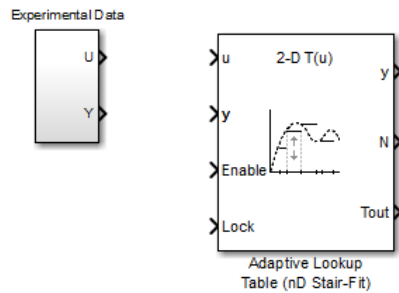**Tip** To learn more, see Adaptive Lookup Table (nD Stair-Fit) block reference page.

After you configure the parameters, the block parameters dialog box looks like the following figure.
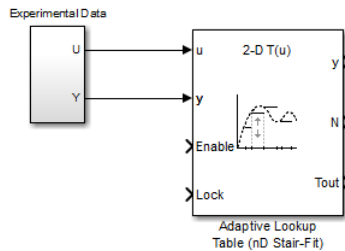
h Click **OK** to close the Function Block Parameters dialog box.

The Simulink model now looks similar to the following figure.

5 Assign the input and output data to the engine model by connecting the U and Y ports of the Experimental Data block to the u and y ports of the Adaptive Lookup Table block, respectively.
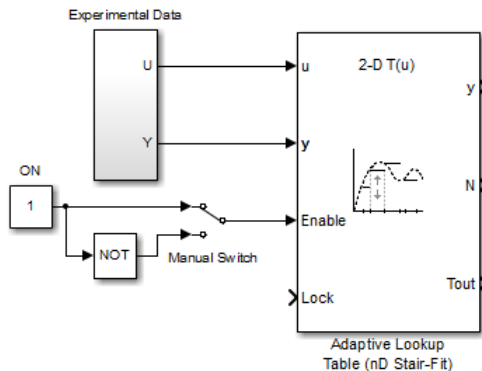

Tip To learn how to connect blocks in the Simulink model window, see "Connect Blocks" in the Simulink documentation.

6 Design a logic using Simulink blocks to enable or disable the adaptation process. Connect the logic to the Adaptive Lookup Table block, as shown in the following figure.


This logic outputs an initial value of 1 which enables the adaptation process.

7 Design a logic to lock a cell during adaptation. Connect the logic to the Adaptive Lookup Table block, as shown in the following figure.
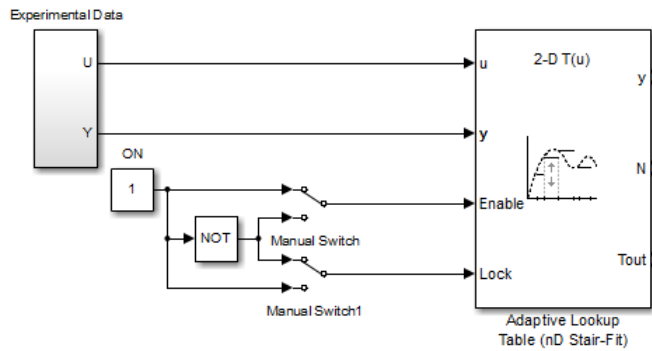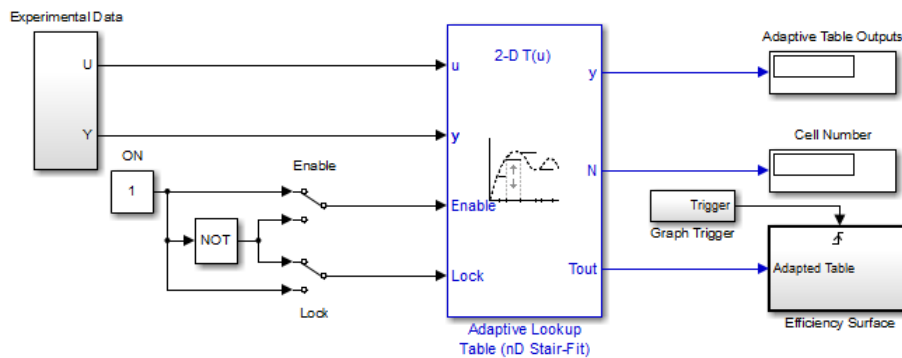
8 In the Simulink Library Browser, select the **Simulink** > **Sinks** library, and drag Display blocks to the model window. Connect the blocks, as shown in the following figure.
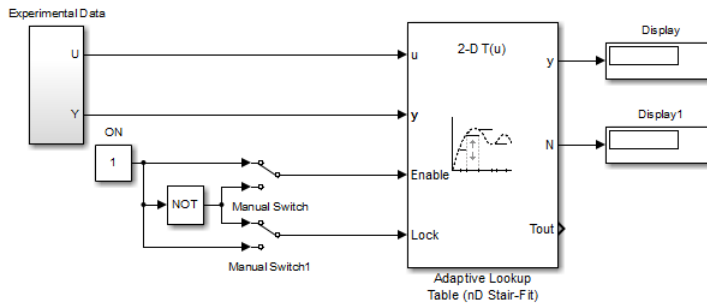
During simulation, the Display blocks show the following:

- `Display` block — Shows the value of the current cell being adapted.
- `Display1` block — Shows the number of the current cell being adapted.

9 Write a MATLAB function to plot the lookup table values as they adapt during simulation.

Alternatively, type `enginetable` at the MATLAB prompt to open a preconfigured Simulink model. The `Efficiency Surface` subsystem contains a function to plot the lookup table values, as shown in the next figure.
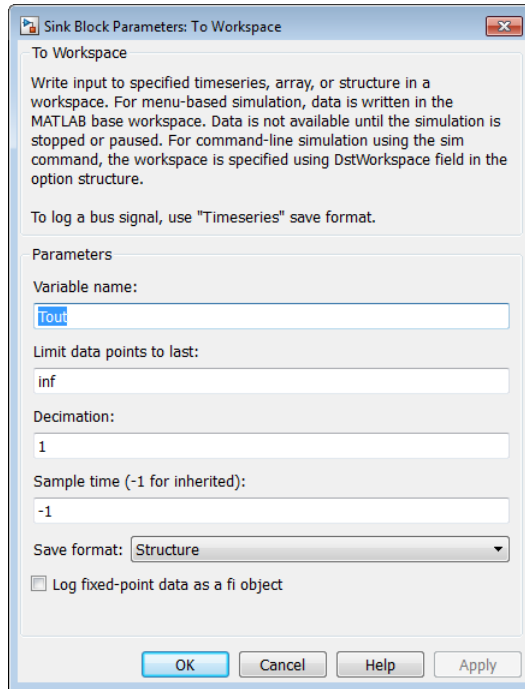
10 Connect a To Workspace block to export the adapted table values:

a In the Simulink Library Browser, select the **Simulink** > **Sinks** library, and drag the To Workspace block to the model window.

To learn more about this block, see the To Workspace block reference page in the Simulink documentation.

b Double-click the To Workspace block to open the Sink Block Parameters dialog box, and type Tout in the **Variable name** field.

c Click **OK**.

d Connect the To Workspace block to the adaptive lookup table output signal `Tout`, as shown in the next figure.
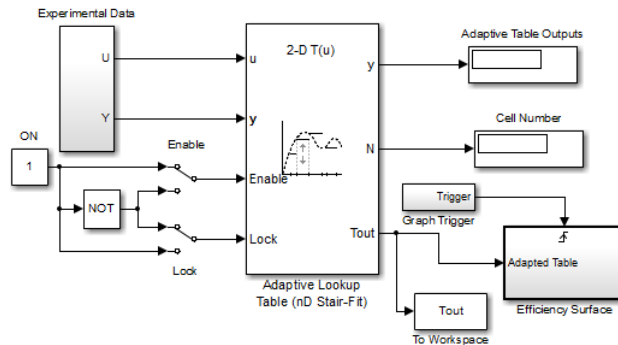
You have now built the model for updating and viewing the adaptive lookup table values. You must now simulate the model to start the adaptation, as described in "Adapting the Lookup Table Values Using Time-Varying I/O Data" on page 6-54.

Adapting the Lookup Table Values Using Time-Varying I/O Data

In this portion of the tutorial, you learn how to update the lookup table values to adapt to the time-varying input and output values.

You must have already built the Simulink model, as described "Building a Model Using Adaptive Lookup Table Blocks" on page 6-45.

To perform the adaptation:

1   In the Simulink Editor, specify the simulation time as `inf`.

    The simulation time of infinity specifies that the adaptation process continues as long as the input and output values of the engine change.

2   In the Simulink Editor, select **Simulation** > **Run** to start the adaptation process.

    A figure window opens that shows the volumetric efficiency of the engine as a function of the intake manifold pressure and engine speed:

    •   The left plot shows the measured volumetric efficiency as a function of intake manifold pressure and engine speed.

    •   The right plot shows the volumetric efficiency as it adapts with the time-varying intake manifold pressure and engine speed.
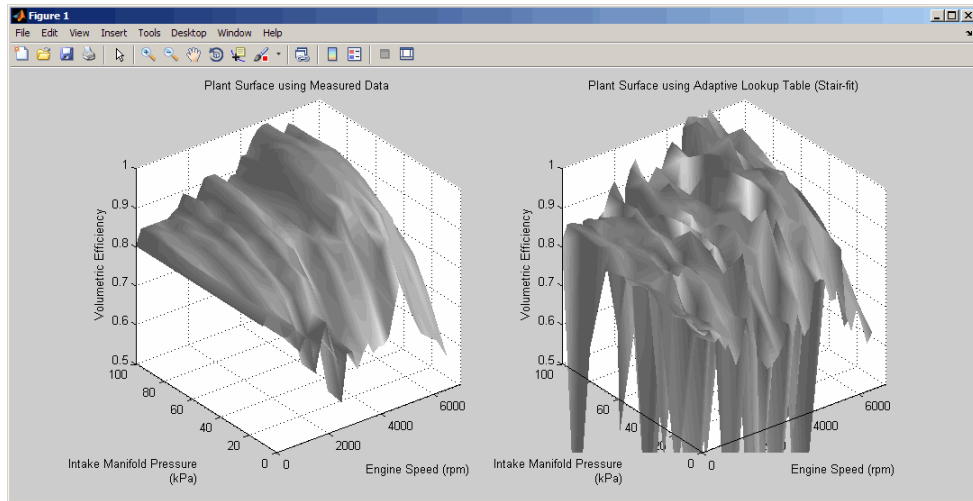
During simulation, the lookup table values displayed on the right plot adapt to the variations in the I/O data. The left and the right plots resemble each other after a few seconds, as shown in the next figure.
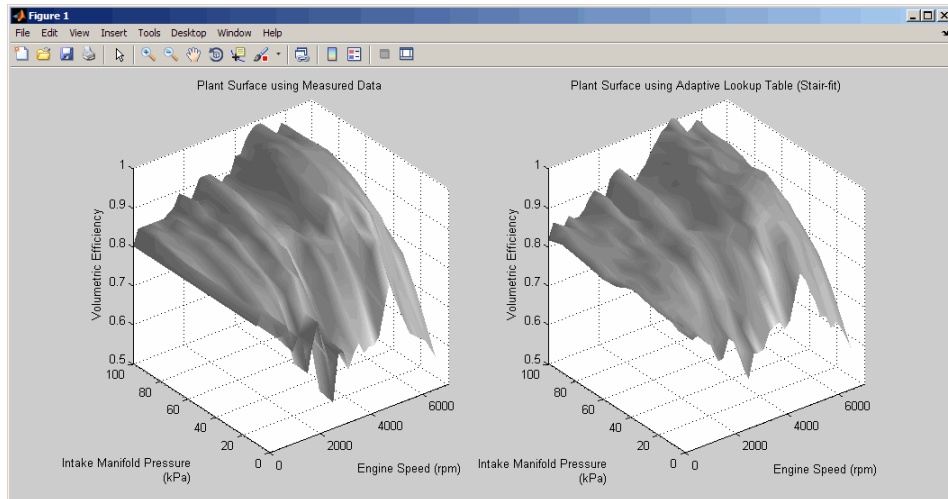
Tip During simulation, the `Cell Number` and `Adaptive Table Outputs` blocks in the Simulink model display the cell number, and the adapted lookup table value in the cell, respectively.

---

3 Pause the simulation by selecting **Simulation > Pause**.

This action also exports the adapted table values `Tout` to the MATLAB workspace.

Note: After you pause the simulation, the adapted table values are stored in the Adaptive Lookup Table block.

---

4 Examine that the left and the right plots match. This resemblance indicates that the table values have adapted to the time-varying I/O data.

5 Lock a table cell so that only one cell adapts. You may find this feature useful if a portion of the data is highly erratic or otherwise difficult for the algorithm to handle.

    a Select **Simulation > Run** to restart the simulation.

    b Double-click the `Lock` block. This action toggles the switch and feeds the output of the `ON` block to the `Lock` input port of the `Adaptive Lookup Table(nD Stair-Fit)` block.

      You can view the number of the locked cell in the `Cell Number` block in the Simulink model.

6 After the table values adapt to the time-varying I/O data, you can continue to use the Adaptive Lookup Table block as a static lookup table:

    a In the Simulink model window, double-click the `Enable` block. This action toggles the switch, and disables the adaptation.

    b Select **Simulation > Run** to restart the simulation, if it is not already running.

      During simulation, the Adaptive Lookup Table block works like a static lookup table, and continues to estimate the output values as the input values change. You can see the current lookup table value in the `Adaptive Table Outputs` block in the Simulink model window.

> **Note:** After you disable the adaptation, the Adaptive Lookup Table block does not update the stored table values, and the figure that displays the table values does not update.

See Also
Adaptive Lookup Table (nD Stair-Fit)

More About
• "What are Adaptive Lookup Tables?" on page 6-2

Using Adaptive Lookup Tables in Real-Time Environment

You can use experimental data from sensor measurements collected by running various tests on a system in real time. The measured data is then sent to the adaptive table block to generate a lookup table describing the relation between the system inputs and output.

You can also use the Adaptive Lookup Table block in a real-time environment, where some time-varying properties of a system need to be captured. To do so, generate C code using Simulink Coder™ code generation software that can then be run in Simulink Real-Time™ or dSPACE® software. Because you can start, stop, or reset the adaptation if you want, use logic to enable the adaptation of the table data only when it is desired. The cell number output N, and the Enable and Lock inputs facilitate this process. Use the Enable input to start and stop the adaptation and the Lock input to update only one of the table cells. The Lock input combined with some logic using the cell number output N provide the means for updating only the desired table cells during a simulation run.

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

Related Examples
- "Model Engine Using n-D Adaptive Lookup Table" on page 6-44

More About
- "What are Adaptive Lookup Tables?" on page 6-2