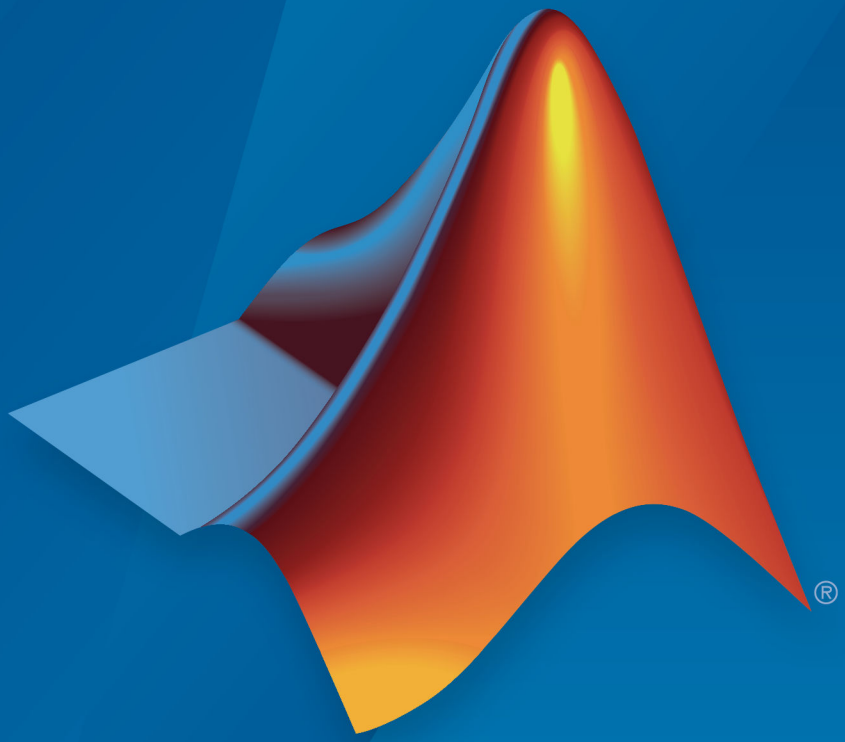
Simulink® Control Design™

User's Guide



MATLAB® & SIMULINK®

R2017b

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

Simulink® Control Design™ User's Guide

Revision History

June 2004	Online only	New for Version 1.0 (Release 14)
October 2004	Online only	Revised for Version 1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.2 (Release 14SP2)
September 2005	Online only	Revised for Version 1.3 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0 (Release 2006a)
September 2006	Online only	Revised for Version 2.0.1 (Release 2006b)
March 2007	Online only	Revised for Version 2.1 (Release 2007a)
September 2007	Online only	Revised for Version 2.2 (Release 2007b)
March 2008	Online only	Revised for Version 2.3 (Release 2008a)
October 2008	Online only	Revised for Version 2.4 (Release 2008b)
March 2009	Online only	Revised for Version 2.5 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 3.4 (Release 2011b)
March 2012	Online only	Revised for Version 3.5 (Release 2012a)
September 2012	Online only	Revised for Version 3.6 (Release 2012b)
March 2013	Online only	Revised for Version 3.7 (Release 2013a)
September 2013	Online only	Revised for Version 3.8 (Release 2013b)
March 2014	Online only	Revised for Version 4.0 (Release 2014a)
October 2014	Online only	Revised for Version 4.1 (Release 2014b)
March 2015	Online only	Revised for Version 4.2 (Release 2015a)
September 2015	Online only	Revised for Version 4.2.1 (Release 2015b)
March 2016	Online only	Revised for Version 4.3 (Release 2016a)
September 2016	Online only	Revised for Version 4.4 (Release 2016b)
March 2017	Online only	Revised for Version 4.5 (Release 2017a)
September 2017	Online only	Revised for Version 5.0 (Release 2017b)

Contents

1 Steady-State Operating Points

2 Linearization

3 Batch Linearization

4 Troubleshooting Linearization Results

5 Frequency Response Estimation

6 PID Controller Tuning

7 PID Autotuning

8 Classical Control Design

9 Control System Tuning

10 Gain-Scheduled Controllers

11 Loop-Shaping Design

12 Model Verification

13 Alphabetical List

14 Blocks — Alphabetical List

15 Objects — Alphabetical List

16 Model Advisor Checks

Steady-State Operating Points

About Operating Points

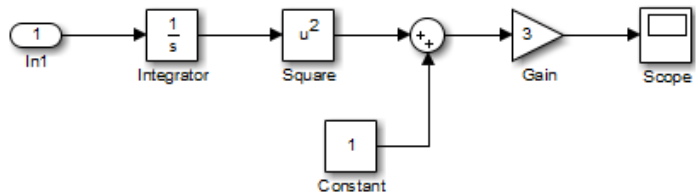
In this section...

What Is an Operating Point?

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

The following Simulink model has an operating point that consists of two variables:

- A root-level input signal set to `1`
- An Integrator block state set to `5`

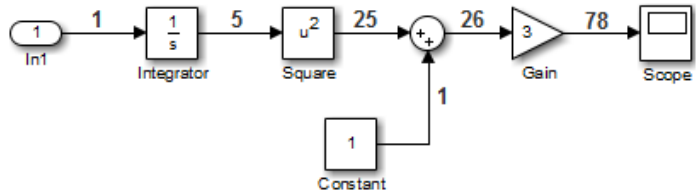


The following table summarizes the signal values for the model at this operating point.

Block	Block Input	Block Operation	Block Output
Integrator	`1`	Integrate input	`5`, set by the initial condition`x0 = 5`
Square	`5`, set by the initial condition of the Integrator block	Square input	`25`

Block	Block Input	Block Operation	Block Output
Sum	`25` from Square block, `1` from Constant block	Sum inputs	`26`
Gain	`26`	Multiply input by 3	`78`

The following block diagram shows how the model input and the initial state of the Integrator block propagate through the model during simulation.



If your model initial states and inputs already represent the desired steady-state operating conditions, you can use this operating point for linearization or control design.

What Is a Steady-State Operating Point?

A *steady-state operating point* of a model, also called an equilibrium or *trim* condition, includes state variables that do not change with time.

A model can have several steady-state operating points. For example, a hanging damped pendulum has two steady-state operating points at which the pendulum position does not change with time. A *stable steady-state operating point* occurs when a pendulum hangs straight down. When the pendulum position deviates slightly, the pendulum always returns to equilibrium. In other words, small changes in the operating point do not cause the system to leave the region of good approximation around the equilibrium value.

An *unstable steady-state operating point* occurs when a pendulum points upward. As long as the pendulum points *exactly* upward, it remains in equilibrium. However, when the pendulum deviates slightly from this position, it swings downward and the operating point leaves the region around the equilibrium value.

When using optimization search to compute operating points for nonlinear systems, your initial guesses for the states and input levels must be near the desired operating point to ensure convergence.

When linearizing a model with multiple steady-state operating points, it is important to have the right operating point. For example, linearizing a pendulum model around the stable steady-state operating point produces a stable linear model, whereas linearizing around the unstable steady-state operating point produces an unstable linear model.

Simulink Model States Included in Operating Point Object

In Simulink Control Design software, an operating point for a Simulink model is represented by an operating point (`operpoint`) object. The object stores the tunable model states and their values, along with other data about the operating point. The states of blocks that have internal representation, such as Backlash, Memory, and Stateflow® blocks, are excluded.

States that are excluded from the operating point object cannot be used in trimming computations. These states cannot be captured with `operspec` or `operpoint`, or written with `initopspec`. Such states are also excluded from operating point displays or computations using Linear Analysis Tool. The following table summarizes which states are included and which are excluded from the operating point object.

State Type	Included in Operating Point?
Double-precision real-valued states .	Yes
States whose value is not of type `double`. For example, complex-valued states, `single`-type states, `int8`-type states.	No
States from root-level inport blocks with double-precision real-valued inputs.	Yes
Internal state representations that impact block output, such as states in Backlash, Memory, or Stateflow blocks.	No (see “Handle Blocks with Internal State Representation” on page 1-86)
States that belong to a Unit Delay block whose input is a bus signal.	No

See Also

`operpoint`

More About

- “Compute Steady-State Operating Points” on page 1-6
- “Handle Blocks with Internal State Representation” on page 1-86
- “Compute Steady-State Operating Point from State Specifications” on page 1-14
- “Compute Steady-State Operating Point from Output Specifications” on page 1-28
- “Compute Operating Points at Simulation Snapshots” on page 1-78

Compute Steady-State Operating Points

In this section...

Steady-State Operating Point Search (Trimming)

You can compute a steady-state operating point (or equilibrium operating point) using numerical optimization methods to meet your specifications. The resulting operating point consists of the equilibrium state values and corresponding model input levels. A successful operating point search finds an operating point very close to a true steady-state solution.

Use an optimization-based search when you have knowledge about the operating point states and the corresponding model input and output signal levels. You can use this knowledge to specify initial guesses or constraints for the following variables at equilibrium:

- Initial state values
- States at equilibrium
- Maximum or minimum bounds on state values, input levels, and output levels
- Known (fixed) state values, input levels, or output levels

Your operating point search might not converge to a steady-state operating point when you *overconstrain* the optimization by specifying:

- Initial guesses for steady-state operating point values that are far away from the desired steady-state operating point.
- Incompatible input, output, or state constraints at equilibrium.

You can control the accuracy of your operating point search by configuring the optimization algorithm settings.

Advantages of Using Simulink Control Design vs. Simulink Operating Point Search

Simulink provides the `trim` command for steady-state operating point searches. However, `findop` in Simulink Control Design provides several advantages over using `trim` when performing an optimization-based operating point search.

	Simulink Control Design Operating Point Search	Simulink Operating Point Search
User interface	Yes	No Only `trim` is available.
Multiple optimization methods	Yes	No Only one optimization method
Constrain state, input, and output variables using upper and lower bounds	Yes	No
Specify the output value of blocks that are not connected to root model outports	Yes	No
Steady-operating points for models with discrete states	Yes	No
Model reference support	Yes	No
Simscape™ Multibody™ integration	Yes	No

Steady-State Operating Point from Simulation Snapshot

You can compute a steady-state operating point by simulating your model until it reaches a steady-state condition. To do so, specify initial conditions for the simulation that are near the desired steady-state operating point.

Use a simulation snapshot when the time it takes for the simulation to reach steady state is sufficiently short. The algorithm extracts operating point values once the simulation reaches steady state.

Simulation-based computations produce poor operating point results when you specify:

- A simulation time that is insufficiently long to drive the model to steady state.
- Initial conditions that do not cause the model to reach true equilibrium.

You can usually combine a simulation snapshot and an optimization-based search to improve your operating point results. For example, simulate your model until it reaches the neighborhood of steady state and use the resulting simulation snapshot to define the initial conditions for an optimization-based search.

Note If your Simulink model has internal states, do not linearize this model at the operating point you compute from a simulation snapshot. Instead, try linearizing the model using a simulation snapshot or at an operating point from optimization-based search.

Which States in the Model Must Be at Steady State?

When computing a steady-state operating point, not all states are required to be at equilibrium. A pendulum is an example of a system where it is possible to find an operating point with all states at steady state. However, for other types of systems, there may not be an operating point where all states are at equilibrium, and the application does not require that all operating point states be at equilibrium.

For example, suppose that you build an automobile model for a cruise control application with these states:

- Vehicle position and velocity
- Fuel and air flow rates into the engine

If your goal is to study the automobile behavior at constant cruising velocity, you need an operating point with the velocity, air flow rate, and fuel flow rate at steady state. However, the position of the vehicle is not at steady state because the vehicle is moving at constant velocity. The lack of a steady-state position variable is fine for the cruise control application because the position does not have significant impact on the cruise control behavior. In this case, you do not need to overconstrain the optimization search for an operating point by requiring that all states be at equilibrium.

Similar situations also appear in aerospace systems when analyzing the dynamics of an aircraft under different maneuvers.

See Also

`findop` | `trim`

More About

- "About Operating Points" on page 1-2
- "Handle Blocks with Internal State Representation" on page 1-86
- "Compute Steady-State Operating Point from State Specifications" on page 1-14
- "Compute Steady-State Operating Point from Output Specifications" on page 1-28
- "Compute Operating Points at Simulation Snapshots" on page 1-78
- "Initialize Steady-State Operating Point Search Using Simulation Snapshot" on page 1-41

View and Modify Operating Points

In this section...

View Model Initial Condition in Linear Analysis Tool

This example shows how to view the model initial condition in the Linear Analysis Tool.

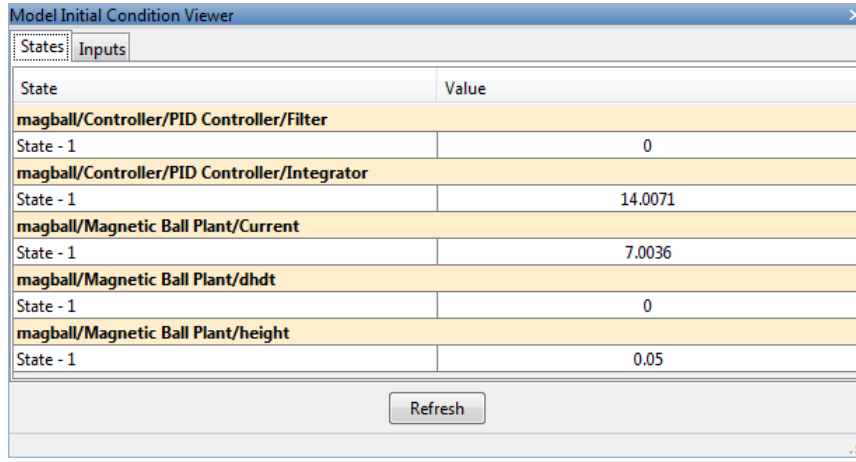
1 Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```

2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

3 In the Linear Analysis Tool, in the **Operating Point** drop-down list, click `View Model Initial Condition`.

This action opens the Model Initial Condition Viewer dialog box, which shows the model initial condition (default operating point).



You cannot edit the `Model Initial Condition` operating point using the Linear Analysis Tool. To edit the initial conditions of the model, change the appropriate

parameter of the relevant block in your Simulink model. For example, double-click the `magball/Magnetic Ball Plant/Current` block to open the Block Parameters dialog box and edit the value in the **Initial condition** box. Click **OK**.

Modify Operating Point in Linear Analysis Tool

This example shows how to modify an existing operating point in the Linear Analysis Tool.

1 Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```

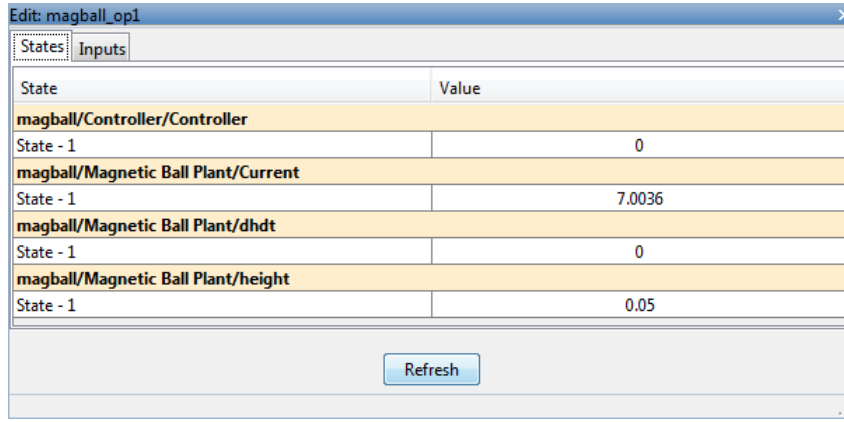
Opening `magball` loads the operating points `magball_op1` and `magball_op2` into the MATLAB® Workspace.

2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

3 In the Linear Analysis Tool, in the **Operating Point** drop-down list, select `magball_op1`.

4 In the **Operating Point** drop-down list, select `Edit magball_op1`.

The Edit dialog box opens for `magball_op1`. Use this dialog box to view and edit the operating point.



Select the state or input **Value** to edit its value.

5 Alternatively, in the Linear Analysis Tool, in the **MATLAB Workspace**, double-click the name of an operating point to open the Edit dialog box.

Note You cannot edit an operating point that you created by trimming a model in the Linear Analysis Tool.

View and Modify Operating Point Object (MATLAB Code)

This example shows how to view and modify the states in a Simulink model using an operating point object.

Create an operating point object from the Simulink Model.

```
sys = 'watertank';
load_system(sys)
op = operpoint(sys)
```

```
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) watertank/PID Controller/Integrator
      x: 0
(2.) watertank/Water-Tank System/H
      x: 1

Inputs: None
----------
```

The operating point, `op`, contains the states and input levels of the model.

Set the value of the first state.

```
op.States(1).x = 1.26;
```

View the operating point state values.

```
op.States
```

```
(1.) watertank/PID Controller/Integrator
      x: 1.26
```

```
(2.) watertank/Water-Tank System/H
      x: 1
```

If you modify your Simulink model after creating an operating point object, then use `update` to update your operating point.

See Also

`operspec` | `update`

More About

- “Simulate Simulink Model at Specific Operating Point” on page 1-83

Compute Steady-State Operating Point from State Specifications

You can compute a steady-state operating point for a Simulink model by specifying constraints on the model states, and finding a model operating condition that satisfies these constraints. For more information on steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-6.

You can trim your model to meet state specifications interactively using the Linear Analysis Tool or programmatically at the MATLAB command line. For each state in your model, you can specify a known value or you can constrain the state value using minimum and maximum bounds. If a state is not known, you can specify an initial guess. You can also specify which states must be at steady-state at the trimmed operating point.

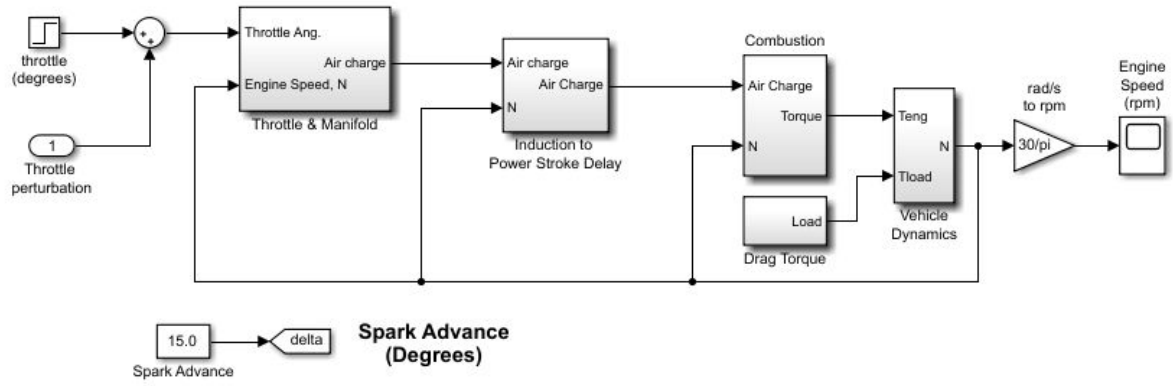
You can also constrain the derivatives of states that are not at steady-state.

Compute Operating Point from State Specifications Using Linear Analysis Tool

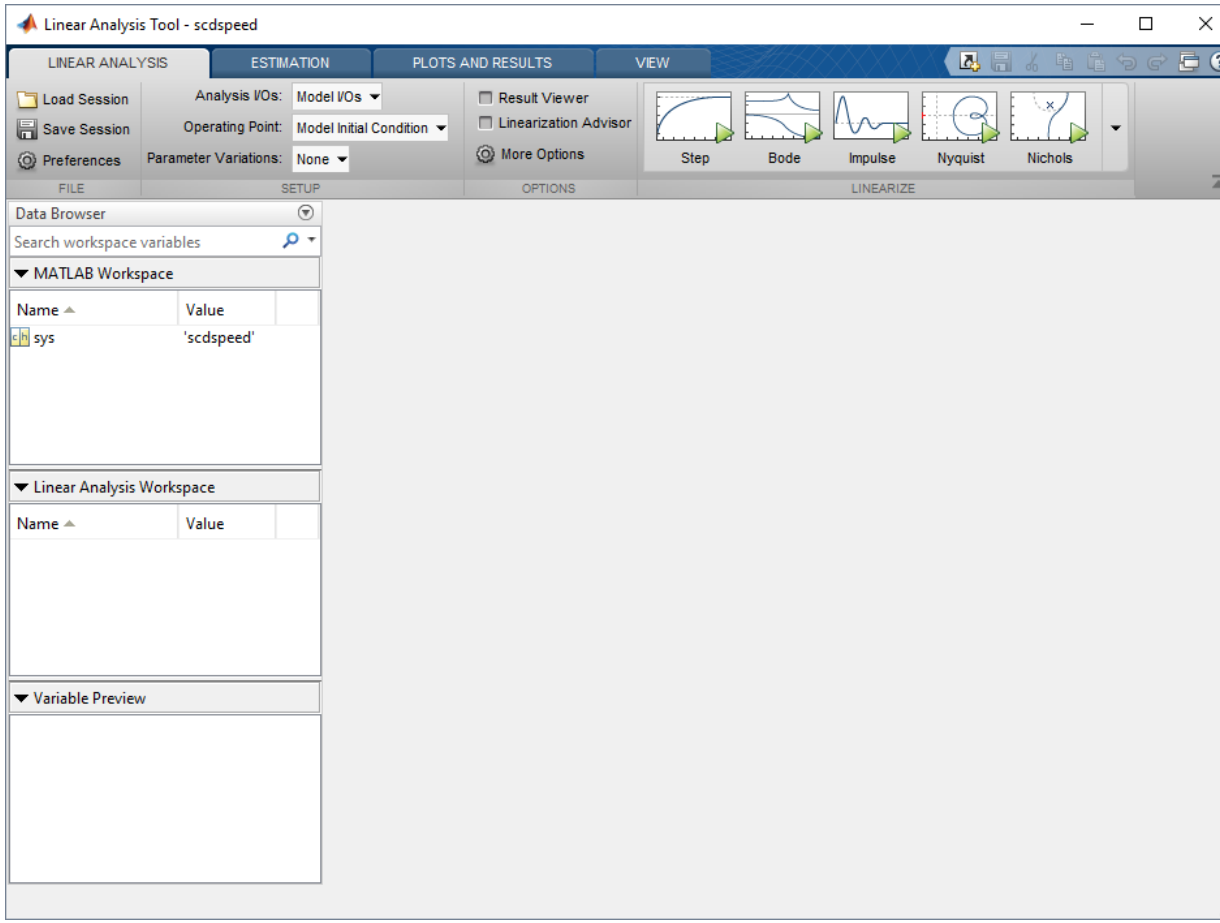
This example shows how to compute a steady-state operating point by specifying known state values and constraints using the Linear Analysis Tool.

Open the Simulink model.

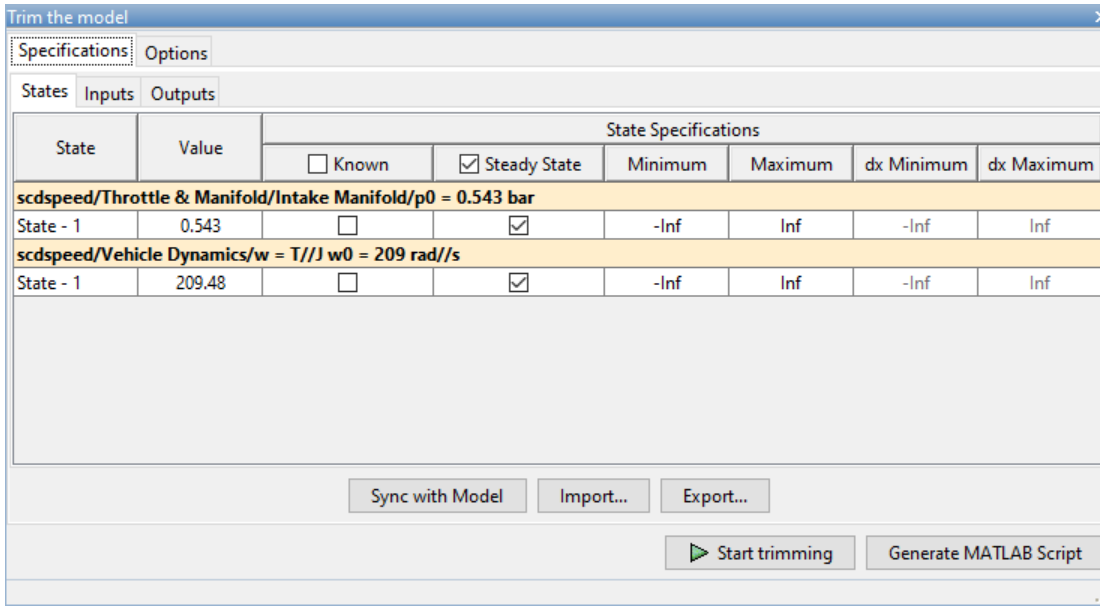
```
sys = 'scdspeed';
open_system(sys)
```



To open the Linear Analysis Tool, in the Simulink model window, select **Analysis** > **Control Design** > **Linear Analysis**.

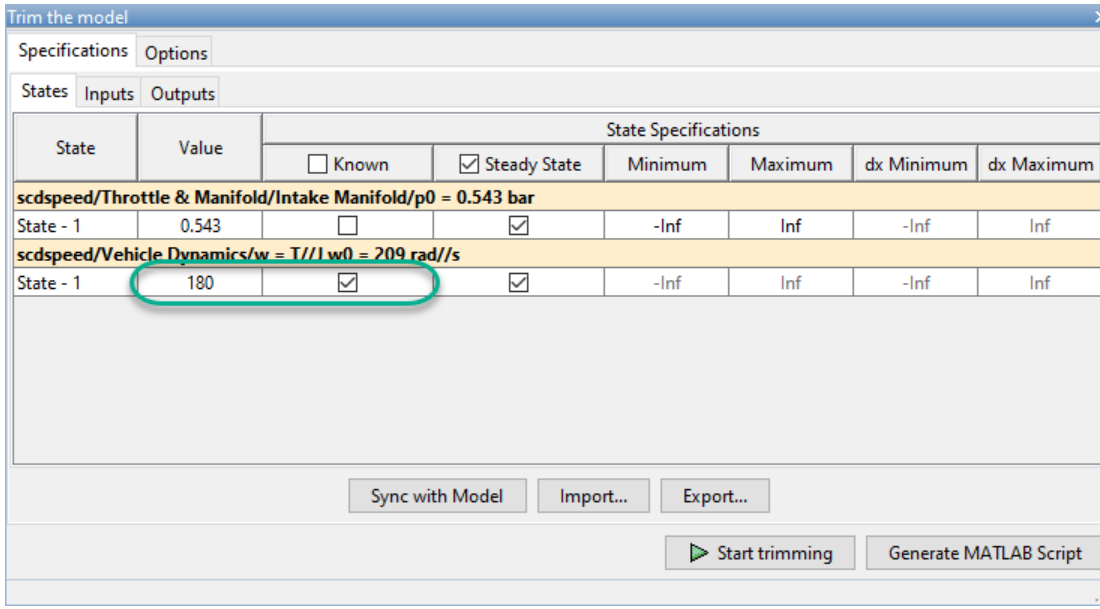


In the Linear Analysis Tool, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Trim Model`.

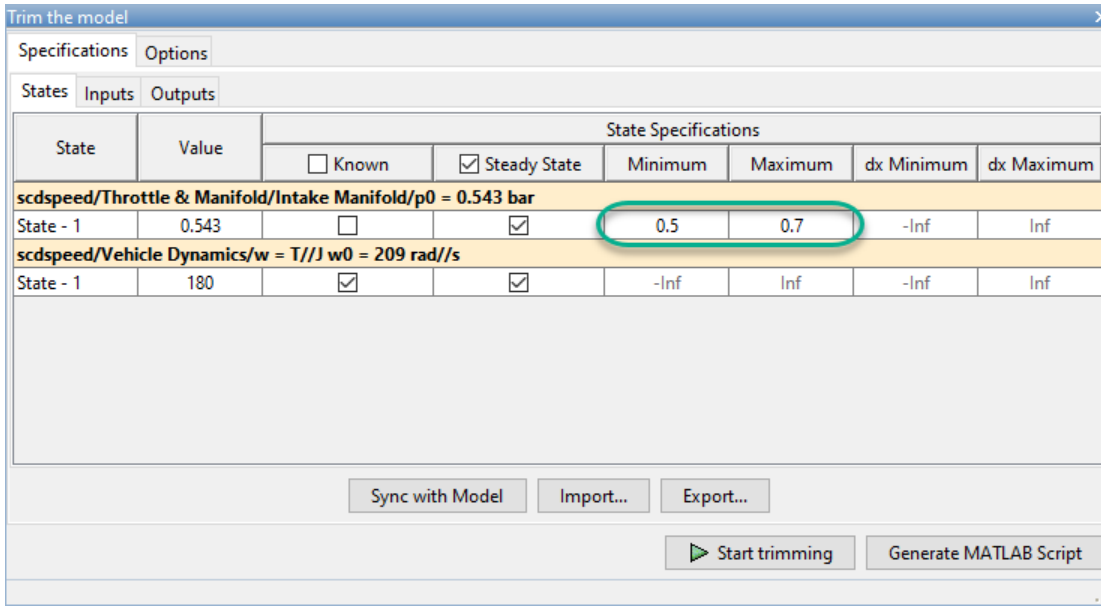


By default, on the **States** tab, the software specifies both model states to be at equilibrium, as shown by the check marks in the **Steady State** column. both states are also specified as unknown values; that is, their steady-state values are calculated during trimming, with an initial guess specified in the **Value** column.

Change the second state, the engine angular velocity, to be a known value. In the **Known** column, select the corresponding row and, in the **Value** column, set the value to `180`.

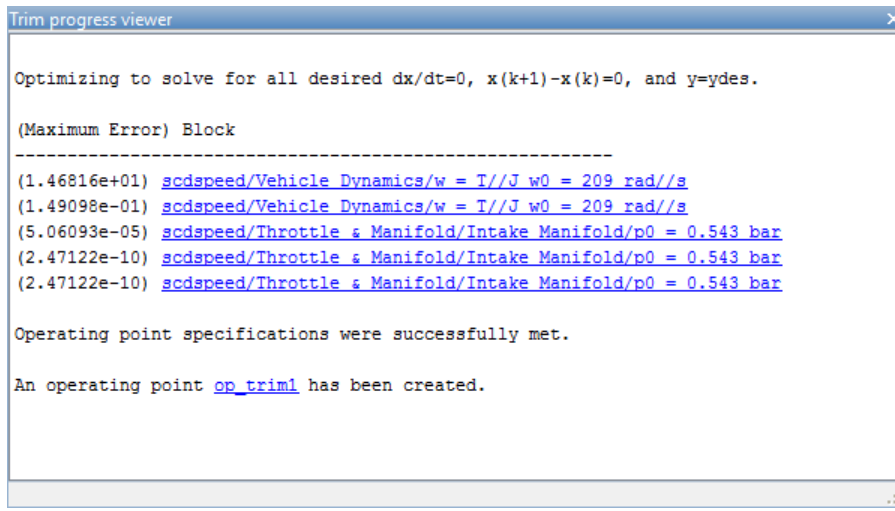


You can also specify bounds for model states during trimming. For this example, constrain the first state to be between `0.5` and `0.7`. To do so, enter these values in the **Minimum** and **Maximum** columns, respectively.



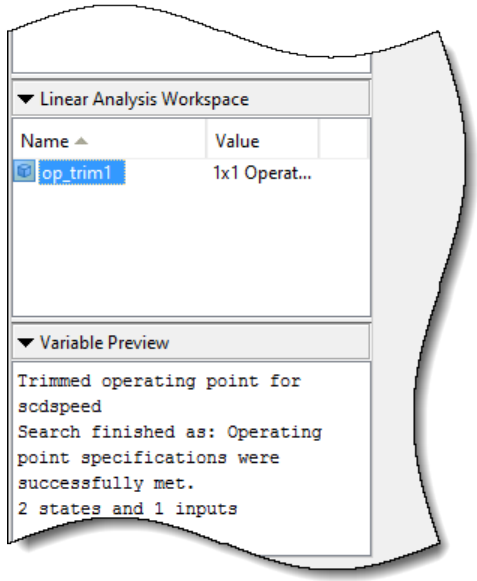
To compute the operating point that meets these specifications, click **Start trimming**.

The software uses optimization to find the operating point that meets your specifications.



The Trim progress viewer shows the optimization progress and that the optimization algorithm terminated successfully. The **(Maximum Error)** column shows the maximum constraint violation at each iteration. The **Block** column shows the block to which the constraint violation applies.

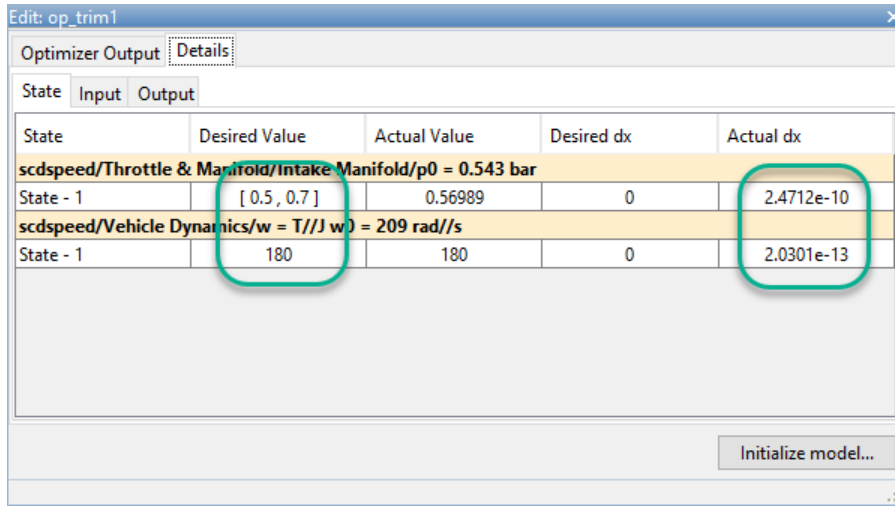
The trimmed operating point, `op_trim1`, appears in the **Linear Analysis Workspace**.



To evaluate whether the resulting operating point values meet the specifications, in the **Linear Analysis Workspace**, double-click `op_trim1`.

In the Edit dialog box, on the **State** tab, the **Actual Value** for the first state falls within the **Desired Value** bounds, and the actual angular velocity is `180`, as specified.

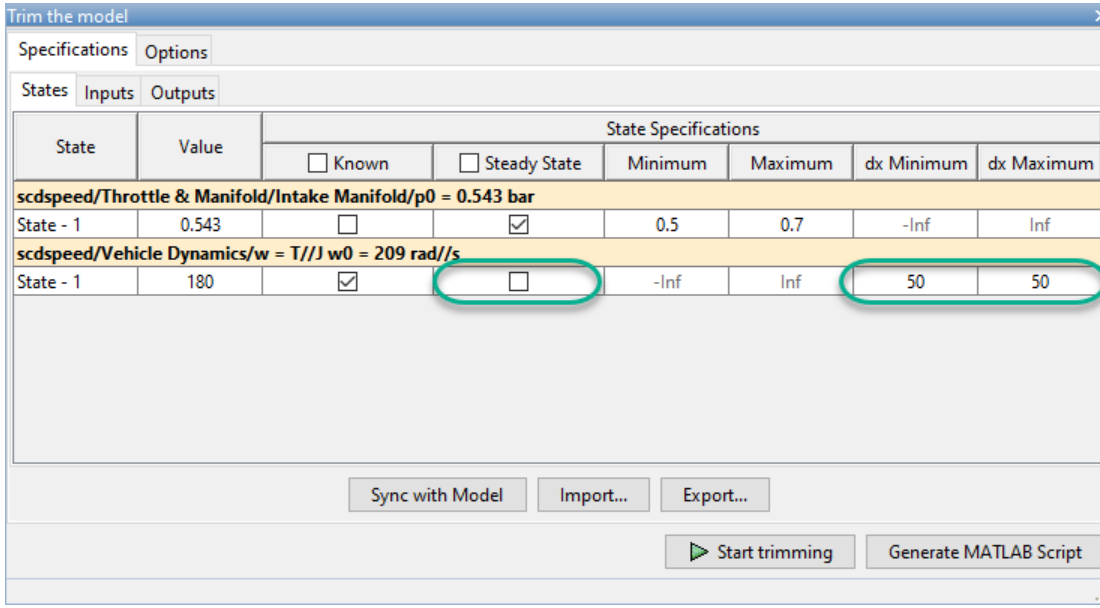
The **Actual dx** column shows the rates of change of the state values at the operating point. Since these values are at or near zero the states are not changing, showing that the operating point is in a steady state.



You can also constrain the derivatives of states that are not at steady state. Using such constraints, you can trim derivatives to known nonzero values or specify derivative tolerances for states that cannot reach steady state.

For example, suppose that you want to find the operating condition at which the engine angular velocity is 180 rad/s and the angular acceleration is `50` rad/s^2. To do so, first open the Trim the model dialog box. In the Linear Analysis Tool, in the **Operating Point** drop-down list, select `Trim Model`.

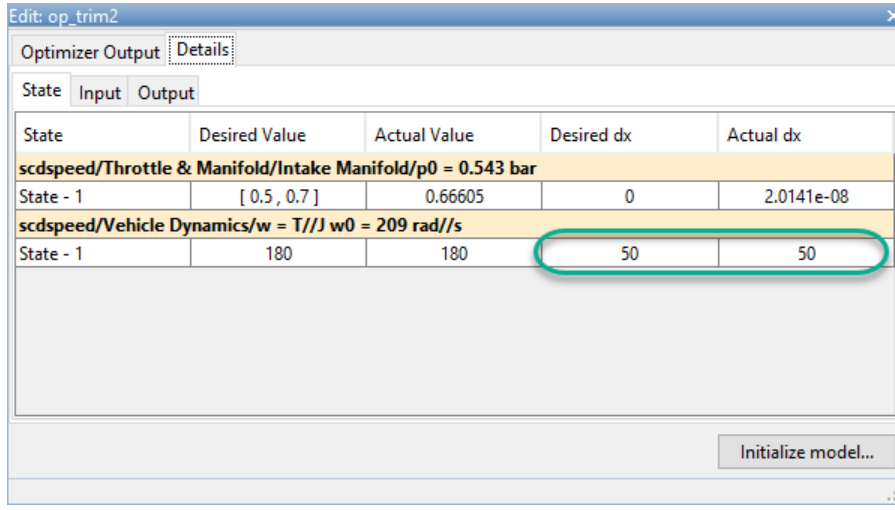
In the **Steady State** column, clear the selection in the corresponding row. Then, in the **dx Minimum** and **dx Maximum** columns, set both state derivative bounds to `50`.



To compute the operating point, click **Start trimming**.

In the Linear Analysis Tool, in the **Linear Analysis Workspace**, double-click `op_trim2`.

In the Edit dialog box, in the second row, the **Actual dx** column matches the **Desired dx** column. Therefore the operating point meets the specified state derivative constraints.

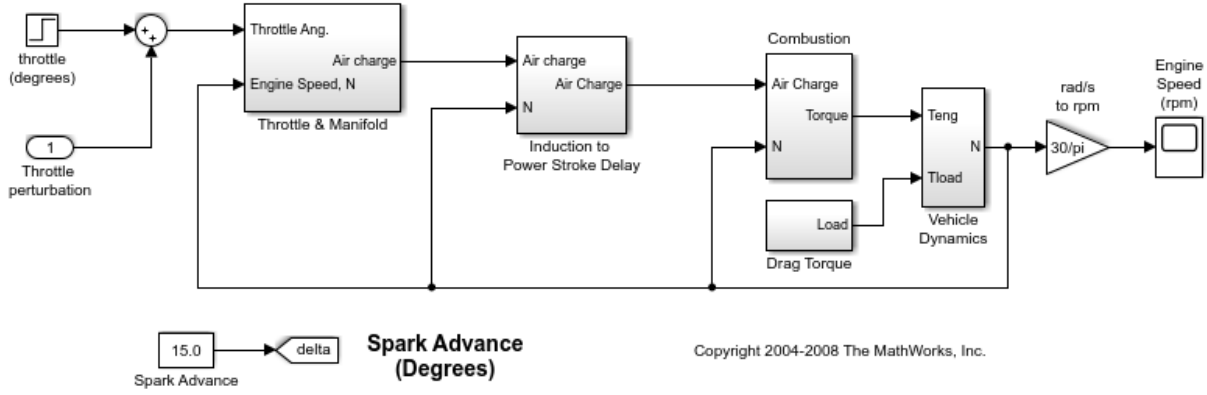


Compute Operating Point from State Specifications at Command Line

This example shows how to compute a steady-state operating point by specifying known state values and constraints.

Open the Simulink model.

```
mdl = 'scdspeed';
open_system(mdl)
```



Create a default operating point specification for the model.

```
opspec = operspec(mdl)
```

```
 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  initial guess: 209

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
     initial guess: 0

Outputs: None
----------
```

By default, both states are specified to be at equilibrium, as shown by the `dx = 0` specification. Both states are also specified as unknown values; that is, their steady-state values are calculated during trimming, with an `initial guess` defined in the specification.

Change the second state, the engine angular velocity, to be a known value, and view the updated state specifications.

```
opspec.States(2).Known = 1;
opspec.States
```

```
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  x:           209
```

The value defined in the second state specification is now the known state value and not an initial guess.

Find an operating point that meets these specifications.

```
op1 = findop(mdl,opspec);


 Operating point search report:
---------------------------------


 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:          0.544      dx:       2.03e-13 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:            209      dx:      -4.57e-13 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:       0.00501    [-Inf Inf]

Outputs: None
----------
```

The operating point search report shows that the specifications were met successfully, and that both states are at steady state as expected (`dx` = 0).

You can also specify bounds for model states during trimming. For example, modify the operating point specifications to trim the second state to a known value of `180`, while constraining the first state to be between `0.5` and `0.7`.

```
opspec.States(2).x = 180;
opspec.States(1).Min = 0.5;
opspec.States(1).Max = 0.7;
```

Find the operating point that meets these specifications.

```
op2 = findop(mdl,opspec);


 Operating point search report:
---------------------------------
```

```
 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:          0.57      dx:      2.47e-10 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           180      dx:      2.03e-13 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:        -0.881    [-Inf Inf]

Outputs: None
----------
```

The operating point search report shows that the specifications were met successfully, and that the first state is within the specified bounds.

Finally, you can also constrain the derivatives of states that are not at steady state. Using such constraints, you can trim derivatives to known nonzero values or specify derivative tolerances for states that cannot reach steady state.

For example, suppose that you want to find the operating condition at which the engine angular velocity is `180` rad/s and the angular acceleration is `50` rad/s^2. To do so, disable the `SteadyState` specification for that state, and set both state derivative bounds to the same target value.

```
opspec.States(2).SteadyState = 0;
opspec.States(2).dxMin = 50;
opspec.States(2).dxMax = 50;
```

The updated state specifications show the new state derivative constraints.

```
opspec.States
```

```
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  50 <= dx <= 50,  x:           180
```

Find an operating point that meets these updated specifications.

```
op3 = findop(mdl,opspec);


 Operating point search report:
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.666      dx:      2.01e-08 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           180      dx:            50 [50, 50]

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:         0.163    [-Inf Inf]

Outputs: None
----------
```

See Also

Apps
Linear Analysis Tool

Functions
`findop` | `operspec`

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Point from Output Specifications” on page 1-28
- “Compute Operating Points at Simulation Snapshots” on page 1-78

- “Change Operating Point Search Optimization Settings” on page 1-46

Compute Steady-State Operating Point from Output Specifications

You can compute a steady-state operating point of a Simulink model by specifying constraints on the model outputs, and finding a model operating condition that satisfies these constraints. For more information on steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-6.

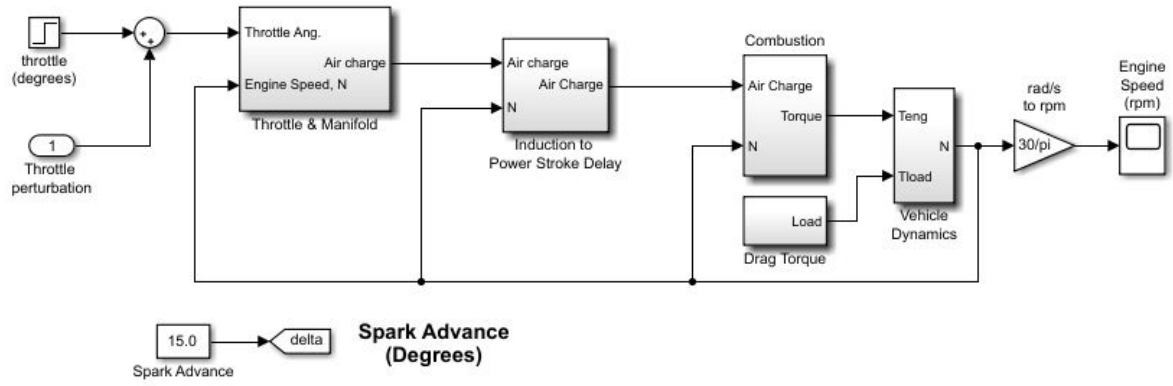
You can trim your model to meet output specifications interactively using the Linear Analysis Tool or programmatically at the MATLAB command line. For each output, you can specify a known value or you can constrain the output value using minimum and maximum bounds. If an output is not known, you can specify an initial guess. You can also specify which outputs must be at steady-state at the trimmed operating point.

Compute Operating Point from Output Specifications Using Linear Analysis Tool

This example shows how to compute a steady-state operating point by specifying known output values and constraints using the Linear Analysis Tool.

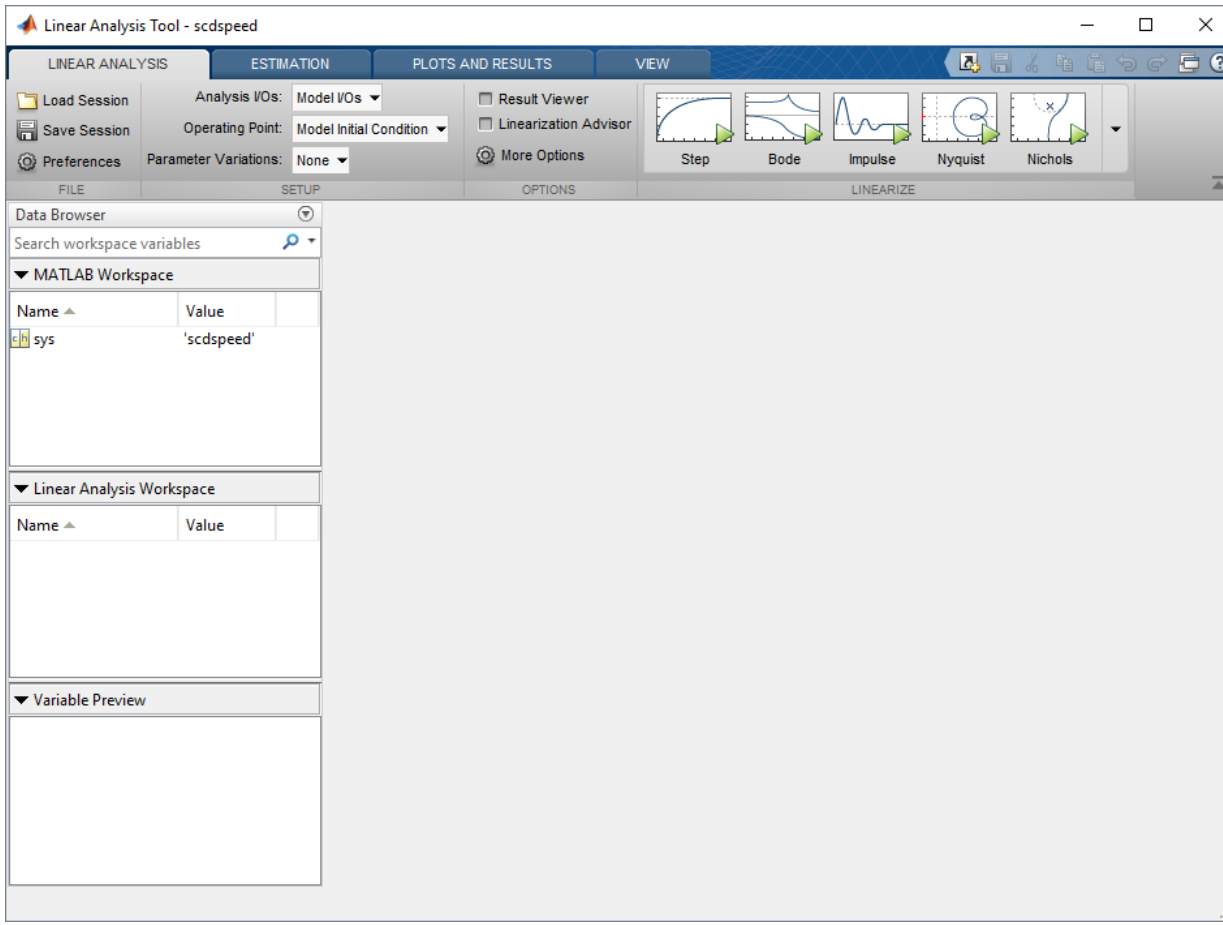
Open the Simulink model.

```
sys = 'scdspeed';
open_system(sys)
```



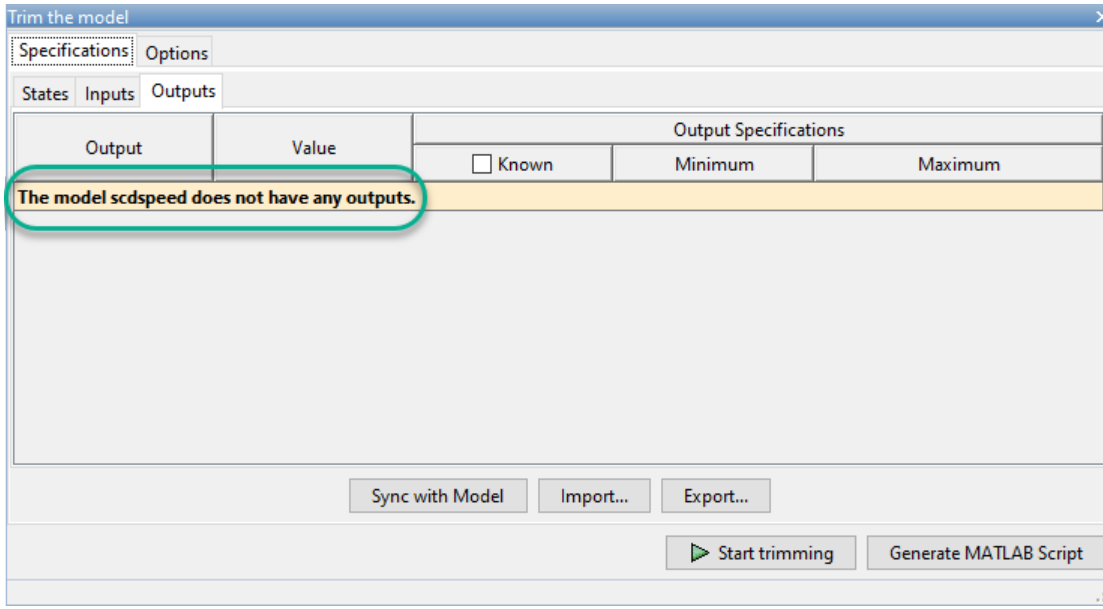
For this example, find a steady-state operating point at which the engine speed is fixed at 2000 rpm.

In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.



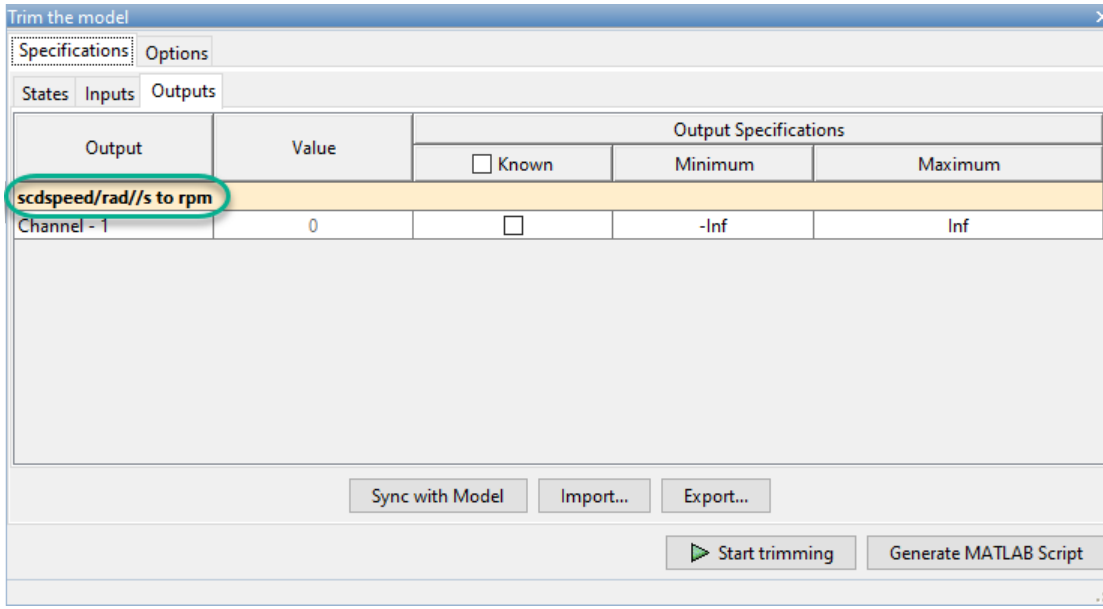
In the Linear Analysis Tool, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Trim Model`.

In the Trim the model dialog box, on the **Outputs** tab, there are no outputs listed since the model has no root-level outputs.

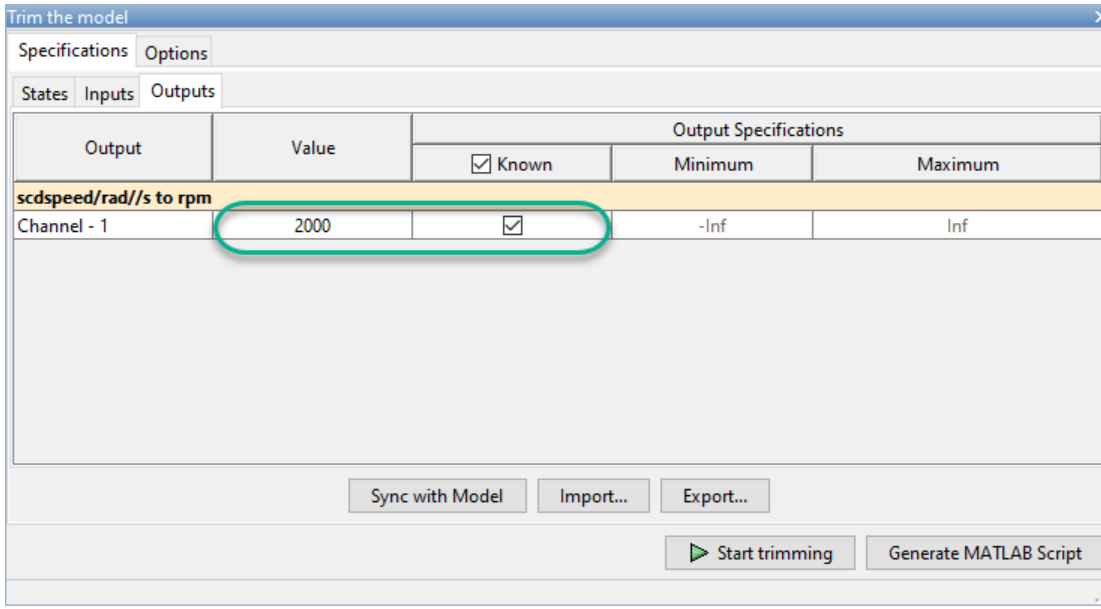


For this example, specify a known steady-state engine speed. To do so, in the Simulink model window, right-click the output signal of the rad/s to rpm block, and select **Linear Analysis Points > Trim Output Constraint**.

The signal constraint marker ⊤ appears in the model, indicating that the signal is available for trimming to an output constraint. The signal now appears in the Trim the model dialog box, under the **Outputs** tab.

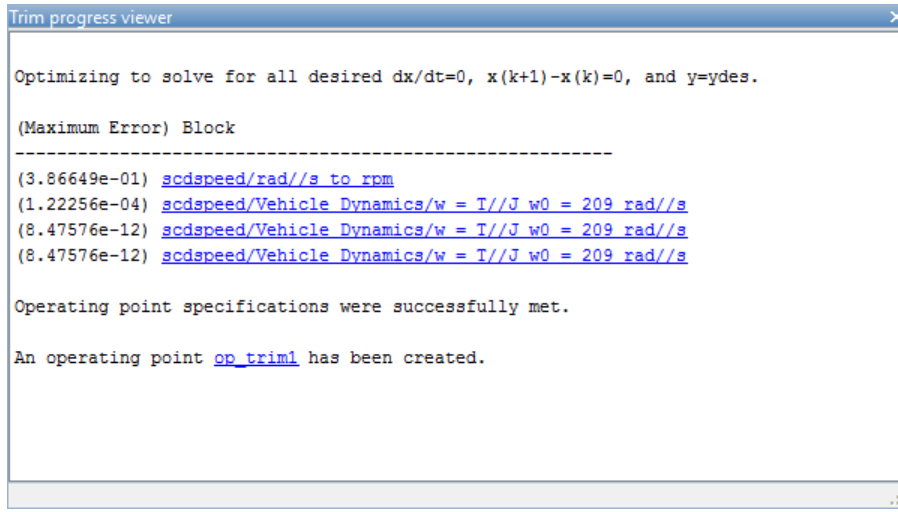


Specify a known speed value. In the **Known** column, select the corresponding row and, in the **Value** column, set the value to `2000`.



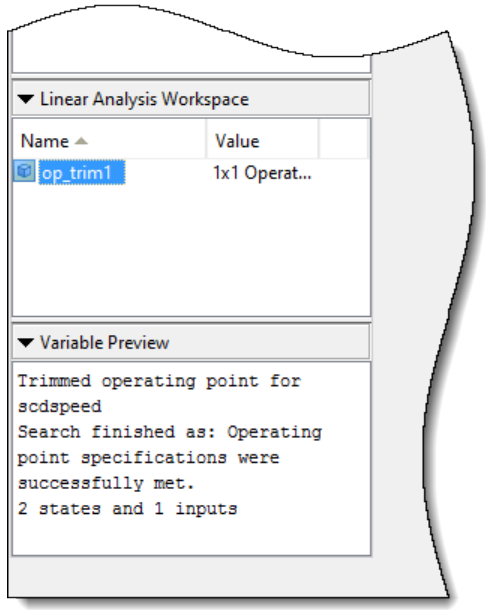
To compute the operating point that meets these specifications, click **Start trimming**.

The software uses optimization to find the operating point that meets your specifications.



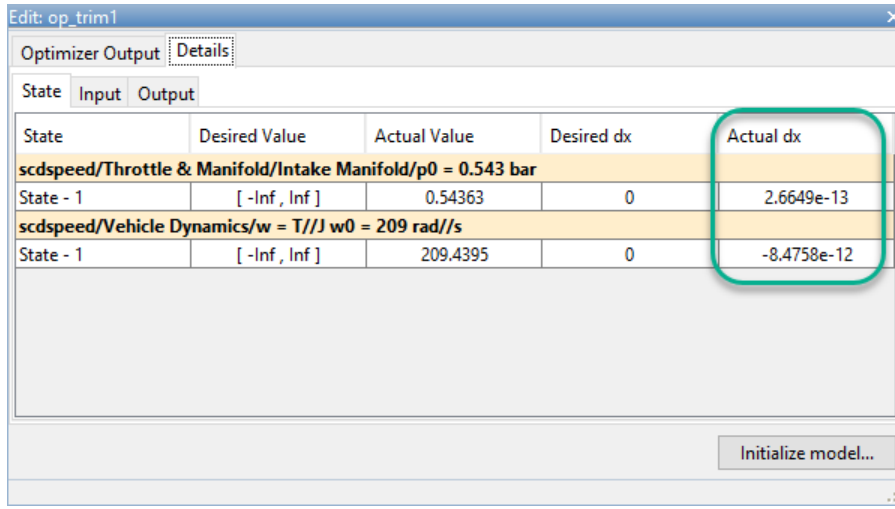
The Trim progress viewer shows the optimization progress and that the optimization algorithm terminated successfully. The **(Maximum Error)** column shows the maximum constraint violation at each iteration. The **Block** column shows the block to which the constraint violation applies.

The trimmed operating point, `op_trim1`, appears in the **Linear Analysis Workspace**.

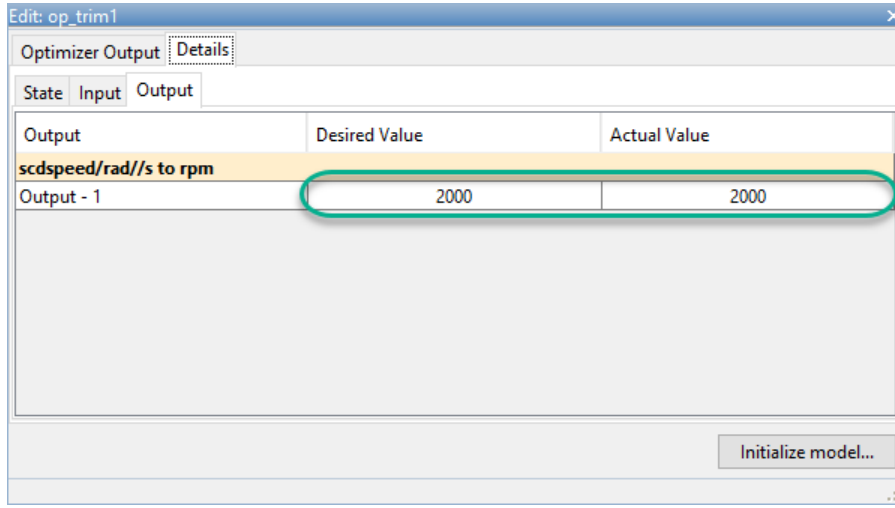


To evaluate whether the resulting operating point values meet the specifications, in the **Linear Analysis Workspace**, double-click `op_trim1`.

In the Edit dialog box, on the **State** tab, the **Actual dx** column shows the rates of change of the state values at the operating point. Since these values are at or near zero, the states are not changing, showing that the operating point is in a steady state.

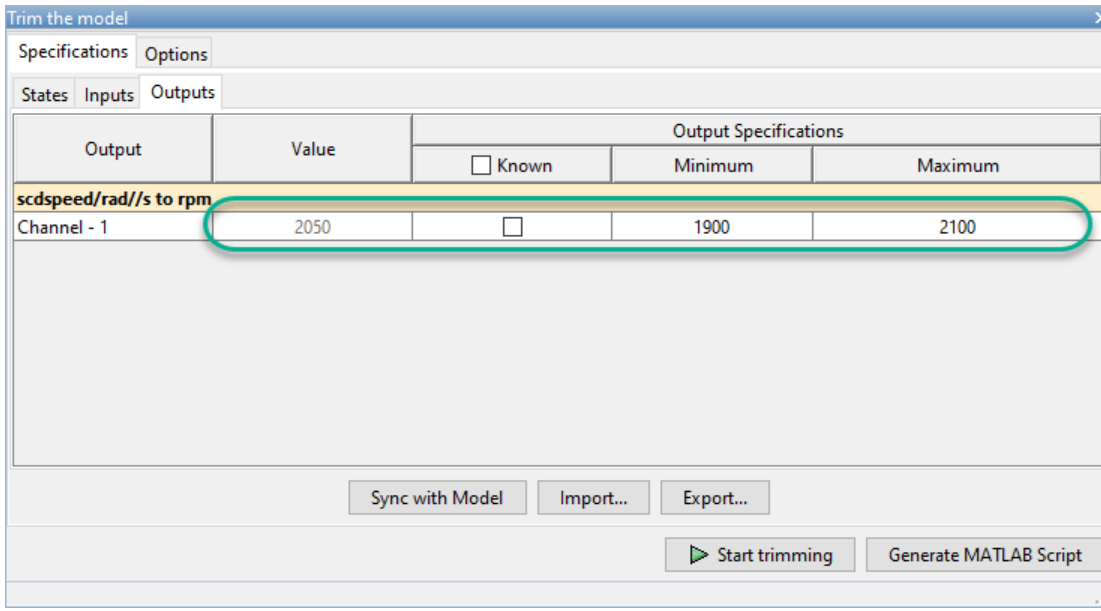


On the **Output** tab, the **Actual Value** and **Desired Value** are both `2000`, showing that the output constraint has been satisfied.



You can also specify bounds for outputs during trimming. For example, suppose that you know that there is a steady-state condition between `1900` and `2100` rpm. To specify this range, in the Trim the model dialog box, on the **Outputs** tab:

- In the **Known** column, clear the entry for the output specification.
- In the **Minimum** and **Maximum** columns, specify the constraint bounds.
- In the **Value** column, specify an initial guess for the value, if you have one.

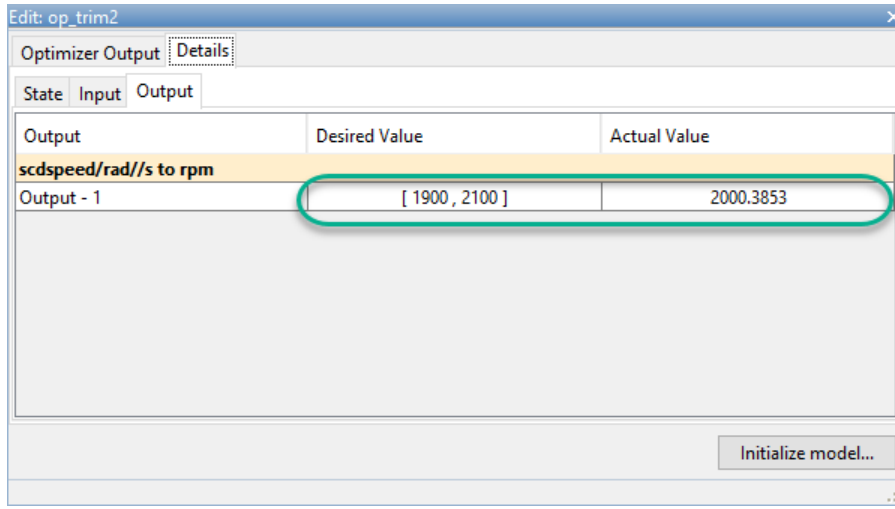


To compute the operating point, click **Start trimming**.

The trimmed operating point, `op_trim2`, appears in the **Linear Analysis Workspace**.

Double-click `op_trim2`.

In the Edit dialog box, on the **Output** tab, the **Actual Value** is within the bounds shown in the **Desired Value** column.

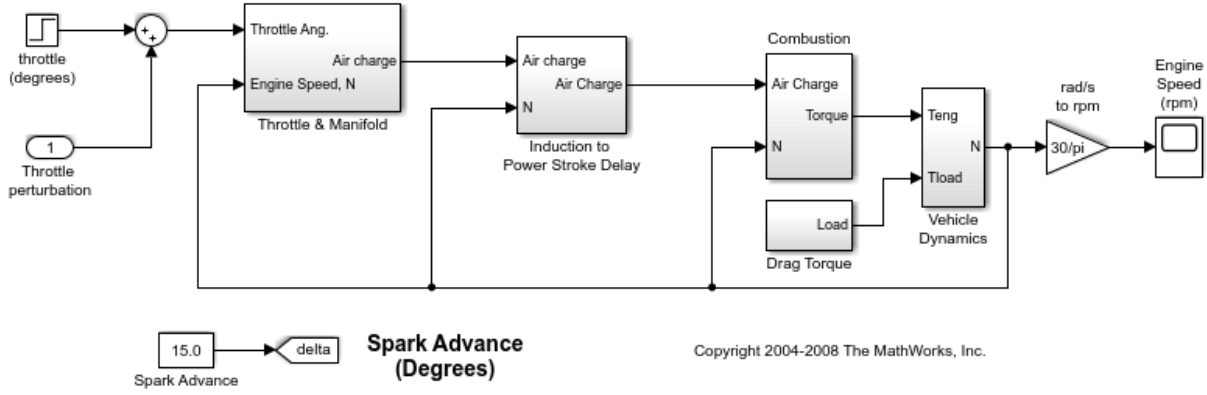


Compute Operating Point from Output Specifications at Command Line

This example shows how to compute a steady-state operating point by specifying known output values and constraints.

Open the Simulink model.

```
mdl = 'scdspeed';
open_system(mdl)
```



Create a default operating point specification for the model.

```
opspec = operspec(mdl)
```

```
 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  initial guess: 209

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
     initial guess: 0

Outputs: None
----------
```

Since there are no root-level outputs in the model, the default operating point specification object has no output specifications.

For this example, specify a known steady-state engine speed. To do so, add an output specification at the output of the rad/s to rpm block.

```
opspec = addoutputspec(opspec,'scdspeed/rad//s to rpm',1);
```

Specify a known value of `2000` rpm for the output constraint.

```
opspec.Outputs(1).Known = 1;
opspec.Outputs(1).y = 2000;
```

View the updated operating point specification.

```
opspec
```

```
 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)
```

```
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  initial guess: 209

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
     initial guess: 0

Outputs:
----------
(1.) scdspeed/rad//s to rpm
     spec:  y = 2e+03
```

Find an operating point that meets these specifications.

```
op1 = findop(mdl,opspec);
```

```
 Operating point search report:
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.544      dx:      2.66e-13 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           209      dx:     -8.48e-12 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:       0.00382    [-Inf Inf]

Outputs:
```

```
----------
(1.) scdspeed/rad//s to rpm
      y:         2e+03    (2e+03)
```

The operating point search report shows that the specifications were met successfully, and that both states are at steady state as expected (`dx` = 0).

You can also specify bounds for outputs during trimming. For example, suppose that you know that there is a steady-state condition between `1900` and `2100` rpm. To trim the speed to this range, modify the operating point specifications.

```
opspec.Outputs(1).Min = 1900;
opspec.Outputs(1).Max = 2100;
```

In this case, since you do not know the output value, specify the output as unknown. You can also provide an initial guess for the output value.

```
opspec.Outputs(1).Known = 0;
opspec.Outputs(1).y = 2050;
```

Find an operating point that meets these specifications.

```
op2 = findop(mdl,opspec);


 Operating point search report:
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.544      dx:      2.99e-13 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           209      dx:      -9.9e-13 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:         0.005    [-Inf Inf]
```

```
Outputs:
----------
(1.) scdspeed/rad//s to rpm
      y:         2e+03    [1.9e+03 2.1e+03]
```

The operating point search report shows that the specifications were met successfully.

See Also

Apps
Linear Analysis Tool

Functions
`addoutputspec` | `findop` | `operspec`

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Point from State Specifications” on page 1-14
- “Compute Operating Points at Simulation Snapshots” on page 1-78
- “Change Operating Point Search Optimization Settings” on page 1-46

Initialize Steady-State Operating Point Search Using Simulation Snapshot

In this section...

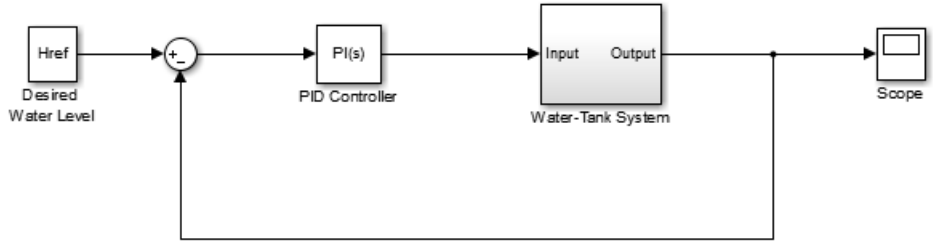
Initialize Operating Point Search Using Linear Analysis Tool

This example shows how to use the Linear Analysis Tool to initialize the values of an operating point search using a simulation snapshot.

If you know the approximate time when the model reaches the neighborhood of a steady-state operating point, you can use simulation to get state values to use as the initial conditions for numerical optimization.

1 Open the Simulink model.

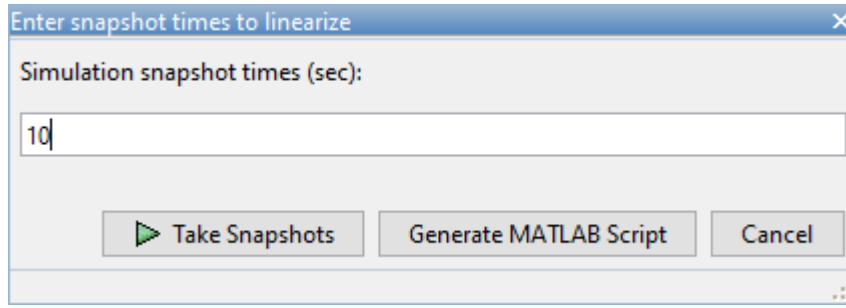
```
sys = ('watertank');
open_system(sys)
```



2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

3 In the Linear Analysis Tool, in the **Operating Point** drop-down list, click `Take Simulation Snapshot`.

4 In the Enter snapshot times to linearize dialog box, enter `10` in the **Simulation snapshot times** field to extract the operating point at this simulation time.



5 Click **Take Snapshots** to take a snapshot of the system at the specified time.

The snapshot, `op_snapshot1`, appears in the **Linear Analysis Workspace** and contains all of the system state values at the specified time.

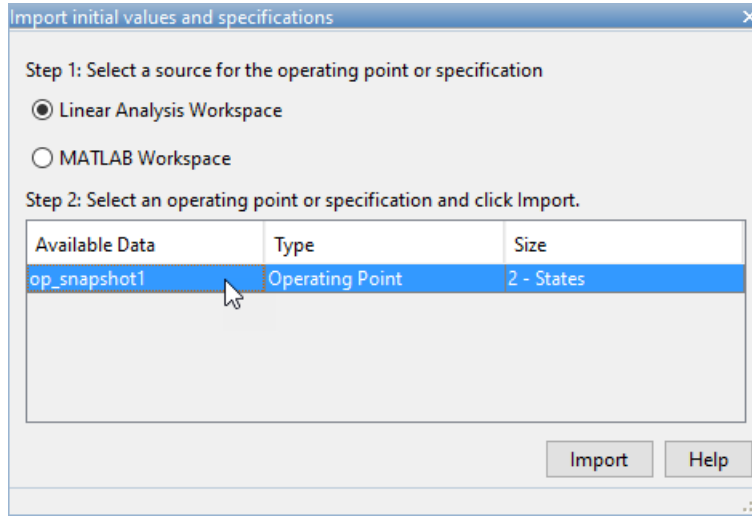
6 In the **Linear Analysis** tab, in the **Operating Point** drop-down list, click `Trim Model`.

The Trim the model dialog box opens.

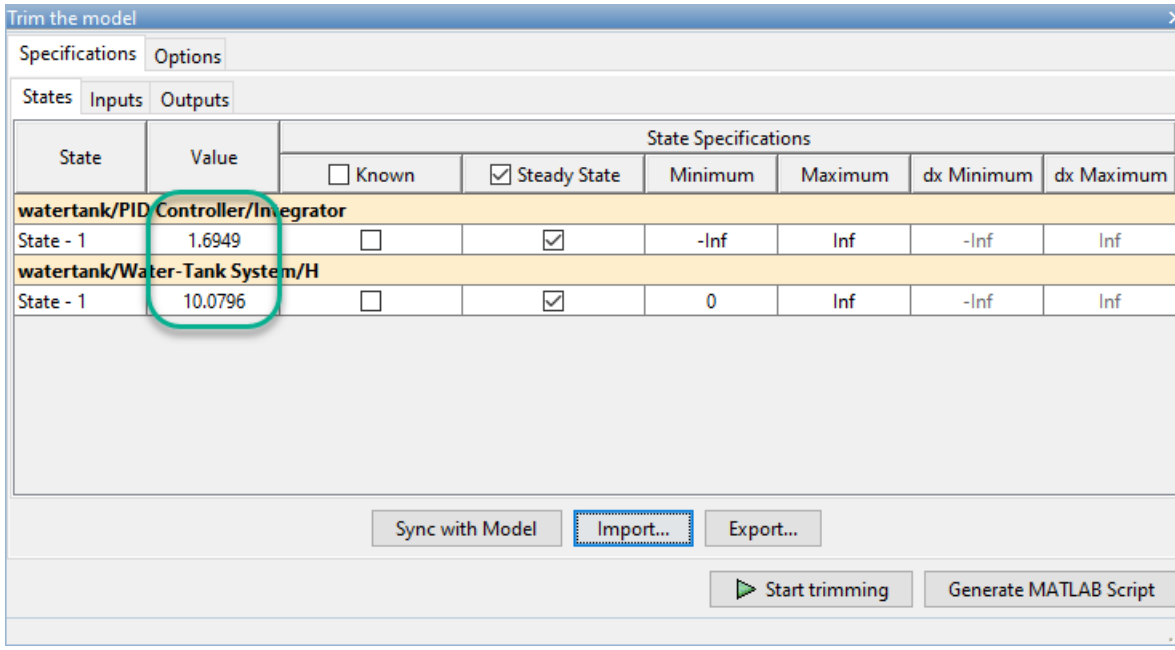
7 Initialize the operating point states with the simulation snapshot values.

Click **Import**.

8 In the Import initial values and specifications dialog box, select `op_snapshot1`, and click **Import**.

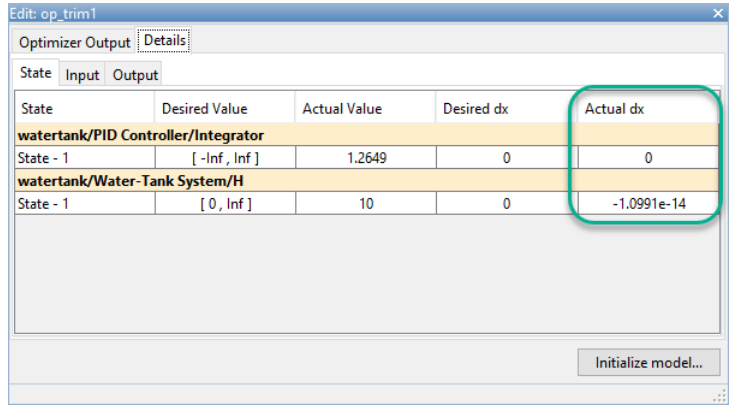


The state values displayed in the Trim the model dialog box update to reflect the imported values.



9 Click **Start trimming** to find the optimized operating point using the states at `t = 10` as the initial values.

10 Double-click `op_trim1` in the **Linear Analysis Workspace** to evaluate whether the resulting operating point values meet the specifications.



The **Actual dx** values are at or near zero, showing that the operating point is at a steady state.

Initialize Operating Point Search Using MATLAB® Code

This example shows how to initialize operating point values for optimization-based operating searches.

Open the Simulink model.

```
sys = 'watertank';
load_system(sys)
```

Simulate the model for ten time units and extract an operating point snapshot.

```
opsim = findop(sys,10);
```

Create an operating point specification object. By default, all model states are specified to be at steady state.

```
opspec = operspec(sys);
```

Configure initial values for operating point search.

```
opspec = initopspec(opspec,opsim);
```

Find the steady-state operating point that meets these specifications.

```
[op,opreport] = findop(sys,opspec);
```

```
 Operating point search report:
---------------------------------

 Operating point search report for the Model watertank.
 (Time-Varying Components Evaluated at time t=10)

Operating point specifications were successfully met.
States:
----------
(1.) watertank/PID Controller/Integrator
      x:          1.26      dx:             0 (0)
(2.) watertank/Water-Tank System/H
      x:            10      dx:      -1.1e-14 (0)
```

```
Inputs: None
----------

Outputs: None
----------
```

The time derivative of each state, `dx`, is effectively zero. This value of the state derivative indicates that the operating point is at steady state.

See Also

`initopspec`

More About

- "Compute Steady-State Operating Points" on page 1-6
- "Change Operating Point Search Optimization Settings" on page 1-46
- "Compute Steady-State Operating Point from State Specifications" on page 1-14

Change Operating Point Search Optimization Settings

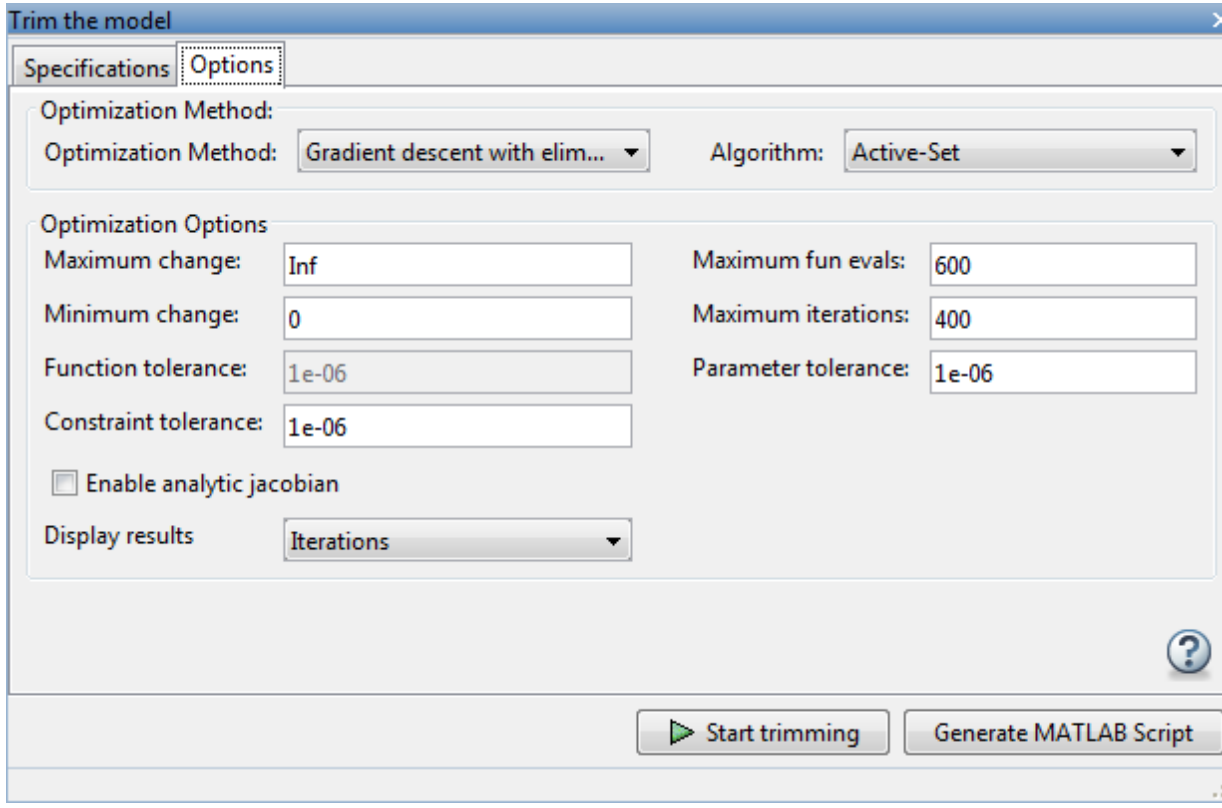
This example shows how to control the accuracy of your operating point search by configuring the optimization algorithm.

Typically, you adjust the optimization settings based on the operating point search report, which is automatically created after each search.

Code Alternative

Use `findopOptions` to configure optimization algorithm settings for `findop`.

1. In the Linear Analysis Tool, open the **Linear Analysis** tab. In the **Operating Point** drop-down list, click **Trim Model**.
2. In the Trim the model dialog box, select the **Options** tab.

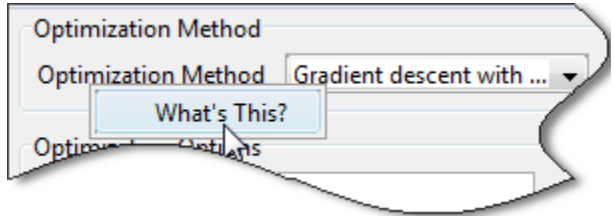


3 Configure your operating point search by selecting an optimization method and changing the appropriate settings.

This table lists the most common optimization settings.

Optimization Status	Option to Change	Comment
Optimization ends before completing (too few iterations)	**Maximum iterations**	Increase the number of iterations
State derivative or error in output constraint is too large	**Function tolerance** or **Constraint tolerance** (depending on selected algorithm)	Decrease the tolerance value

Note You can get help on each option by right-clicking the option label and selecting **What's This?**.



See Also

More About

- "Compute Steady-State Operating Points" on page 1-6
- "Compute Steady-State Operating Point from State Specifications" on page 1-14
- "Compute Steady-State Operating Point from Output Specifications" on page 1-28
- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-75

Import and Export Specifications For Operating Point Search

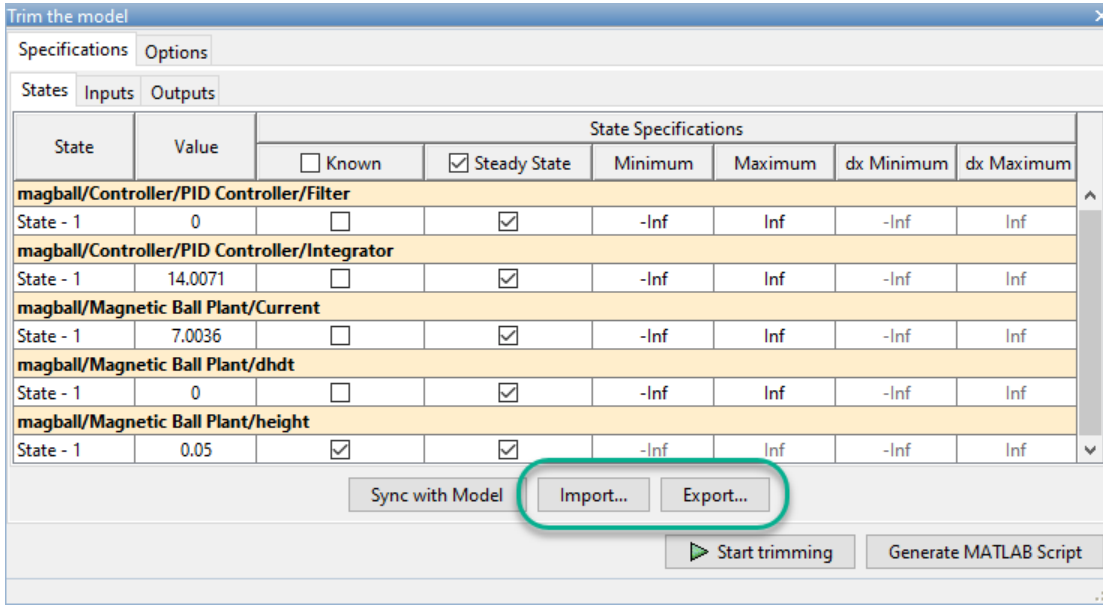
When you modify an operating point specification in the Linear Analysis Tool, you can export the specification to the MATLAB workspace or the Linear Analysis Tool workspace. Exported specifications are saved as operating point specifications objects (see `operspec`). Exporting specifications can be useful when you expect to perform multiple trimming operations using the same or a very similar set of specifications. Additionally, you can export interactively-edited operating point specifications when you want to use the `findop` command to perform multiple trimming operations with a single compilation of the model. (See "Batch Compute Steady-State Operating Points for Multiple Specifications" on page 1-61.)

You can also import saved operating point specifications to the Linear Analysis Tool and use them to interactively compute trimmed operating points. Importing a specification can be useful when you want to trim a model to a specification that is similar to one you previously saved. In that case, you can import the specification to the Linear Analysis Tool and interactively change it. You can then export the modified specification, or compute a trimmed operating point from it.

To import or export an operating point specification:

- In the Linear Analysis Tool, on the **Linear Analysis** tab, select `Trim Model` from the **Operating Point** drop-down list.

  The Trim the model dialog box opens.



- Click **Import** to load a saved operating point specification from the Linear Analysis Workspace or the MATLAB Workspace.
- Click **Export** to save an operating point specification to the Linear Analysis Workspace or the MATLAB Workspace.

For more information about operating point specifications, see the `operspec` and `findop` reference pages.

See Also

`findop` | `operspec`

More About

- “View and Modify Operating Points” on page 1-10
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61

Compute Operating Points Using Custom Constraints and Objective Functions

Typically, when computing a steady-state operating point using an optimization-based search, you specify known fixed values or bounds to constrain your model states, inputs, or outputs. However, some systems or applications require additional flexibility in defining the optimization search parameters.

For such systems, you can specify custom constraints, an additional optimization objective function, or both. When the software computes a steady-state operating point, it applies these custom constraints and objective function in addition to the standard state, input, and output specifications. The following example shows how to define custom constraints and a custom objective function for a Simulink model.

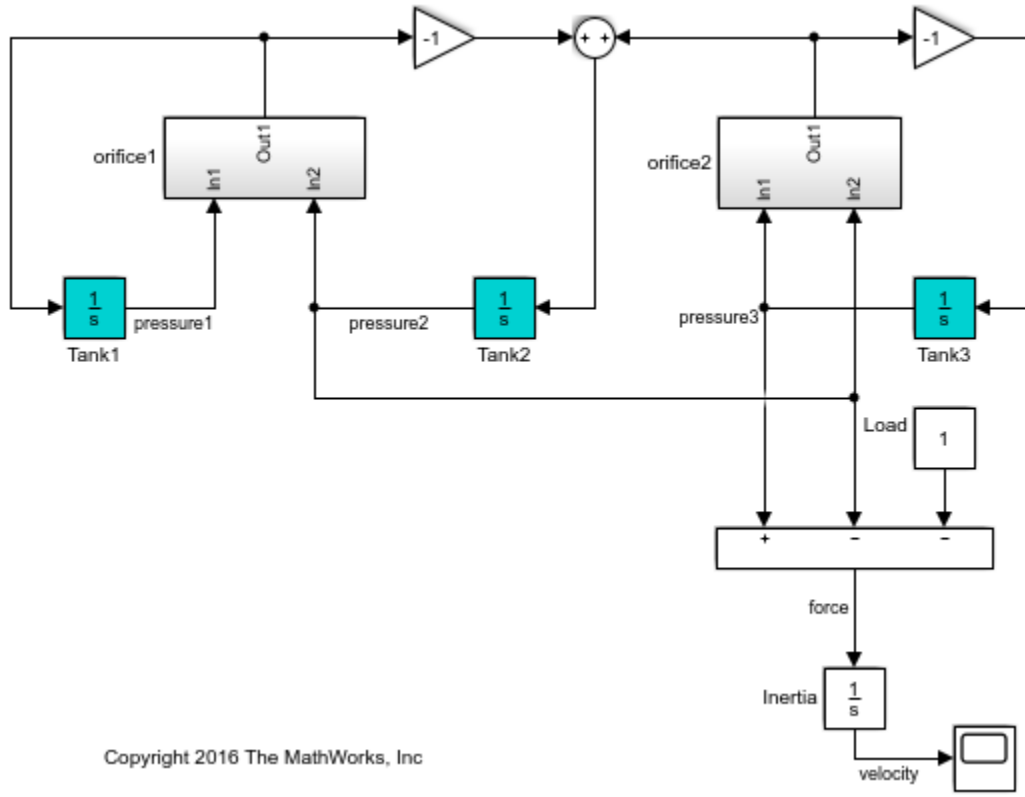
You can specify custom equality and inequality constraints as algebraic combinations of model states, inputs, and outputs. These constraints let you limit the operating point search space by specifying known relationships between inputs, outputs, and states. For example, you can specify that one model state is the sum of two other states.

You can also specify a custom scalar objective function as an algebraic combination of model states, inputs, and outputs. Using the objective function you can optimize the steady-state operating point based on your application requirements. For example, suppose that your model has multiple potential equilibrium points. You can specify an objective function to find the steady-state point with the minimum input energy.

Simulink Model

For this example, use a model of three tanks connected with each other by orifices.

```
mdl = 'scdTanks';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



The flow between Tank1 and Tank2 is desired. The flow between Tank2 and Tank3 is undesired unavoidable leakage.

At the expected steady state of this system:

- Tank1 and Tank2 have the same pressure.
- Tank2 and Tank3 have an almost constant pressure difference of `1` that compensates a load.

Due to the weak connectivity between Tank1 and Tank2, it is difficult to trim the model such that the pressures in Tank1 and Tank2 are equal.

Trim Model Without Customizations

Create a default operating point specification for the model. The specification configures all three tank pressures as free states that must be at steady state in the trimmed operating point.

```
opspec = operspec(mdl);
```

Create an option set for trimming the model, suppressing the Command Window display of the operating point search report. The specific trimming options depend on your application. For this example, use nonlinear least squares optimization.

```
opt = findopOptions('OptimizerType','lsqnonlin');
opt.DisplayReport = 'off';
```

Trim the model, and view the trimmed tank pressures.

```
[op0,rpt0] = findop(mdl,opspec,opt);
op0.States
```

```
(1.) scdTanks/Inertia
      x: 0
(2.) scdTanks/Tank1
      x: 9
(3.) scdTanks/Tank2
      x: 9.5
(4.) scdTanks/Tank3
      x: 10.5
```

The trimmed pressures in Tank1 and Tank2 do not match. Thus, the default operating point specification fails to find an operating point that meets the expected steady-state requirements. If you reduce the constraint tolerance, `opt.OptimizationOptions.TolCon`, you cannot achieve a feasible steady-state solution due to the leakage between Tank2 and Tank3.

Add Custom Constraints

To specify custom constraints, define a function in the current working folder or on the MATLAB path with input arguments:

- `x` - Operating point specification states, specified as a vector.
- `u` - Operating point specification inputs, specified as a vector.

- `y` - Operating point specification outputs, specified as a vector.

and output arguments:

- `c_ineq` - Inequality constraints which must satisfy `c_ineq <= 0` during trimming, returned as a vector.
- `c_eq` - Equality constraints which must satisfy `c_eq = 0` during trimming, returned as a vector.

Each element of `c_ineq` and `c_eq` specifies a single constraint. Define the specific constraints for your application as algebraic combinations of the states, inputs, and outputs. If there are no custom equality or inequality constraints, return the corresponding output argument as `[]`.

For this example, to satisfy the conditions of the expected steady state, define the following custom constraint function.

```
function [c_ineq,c_eq] =  myConstraints(x,u,y)
    c_ineq = [];
    c_eq = [x(2)-x(3);      % Tank1 pressure - Tank2 pressure
            x(3)-x(4)+1];   % Tank2 pressure - Tank3 pressure + 1
end
```

The first entry of `c_eq` constrains the pressures of Tank1 and Tank2 to be the same value. The second equality constraint defines the pressure drop between Tank2 and Tank3.

Add the custom constraint function to the operating point specification.

```
opspec.CustomConstrFcn = @myConstraints;
```

Trim the model using the revised operating point specification that contains the custom constraints, and view the trimmed state values.

```
[op1,rpt1] = findop(mdl,opspec,opt);
op1.States
```

```
(1.) scdTanks/Inertia
      x: 0
(2.) scdTanks/Tank1
      x: 9.33
```

```
(3.) scdTanks/Tank2
      x: 9.33
(4.) scdTanks/Tank3
      x: 10.3
```

Trimming the model with the custom constraint function produces an operating point with equal pressures in the first and second tanks, as expected. Also, as expected, there is a pressure differential of `1` between the third and second tanks.

To examine the final values of the specified constraints, you can check the `CustomEqualityConstr` and `CustomInequalityConstr` properties of the operating point search report.

```
rpt1.CustomEqualityConstr
```

```
ans =

   1.0e-06 *

   -0.0001
   -0.1540
```

The near-zero values indicate that the equality constraints are satisfied.

Add Custom Objective Function

To specify a custom objective function, define a function with the same input arguments as the custom constraint function (`x`, `u`, and `y`), and output argument `F`. `F` is an objective function value to be minimized during trimming, returned as a scalar.

Define the objective function for your application as an algebraic combination of the states, inputs, and outputs.

For this example, assume that you want to keep the pressure in Tank3 in the range [16,20]. However, this condition is not always feasible. Therefore, rather than impose hard constraints, add an objective function to incur a penalty if the pressures are not in the [16,20] range. To do so, define the following custom objective function.

```
function F =  myObjective(x,u,y)
    F = max(x(4)-20, 0) + max(16-x(4), 0);
```

```
end
```

Add the custom objective function to the operating point specification object.

```
opspec.CustomObjFcn = @myObjective;
```

Trim the operating point using both the custom constraints and the custom objective function, and view the trimmed state values.

```
[op2,rpt2] = findop(mdl,opspec,opt);
op2.States
```

```
(1.) scdTanks/Inertia
      x: 0
(2.) scdTanks/Tank1
      x: 15
(3.) scdTanks/Tank2
      x: 15
(4.) scdTanks/Tank3
      x: 16
```

In the trimmed operating point, the pressure in Tank3 is within the [16,20] range specified in the custom objective function.

To view the final value of the scalar objective function, check the `CustomObj` property of the operating point search report.

```
rpt2.CustomObj
```

```
ans =

     0
```

Add Custom Mapping

For complex models, you can define a custom mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. Doing so simplifies the constraint and objective functions by eliminating unneeded states, inputs, and outputs.

To specify a custom mapping, define a function with your operating point specification, `opspec`, as an input argument, and output arguments:

- `indx` - Indices of mapped states
- `indu` - Indices of mapped inputs
- `indy` - Indices of mapped outputs

To obtain state, input, and output indices based on block paths and state names use `getStateIndex`, `getInputIndex`, and `getOutputIndex`. Using these commands is robust to future model changes, such as the addition of model states. Alternatively, you can manually specify the indices. For more information on the format of `indx`, `indu`, and `indy`, see `getStateIndex`, `getInputIndex`, and `getOutputIndex`.

If there are no states, inputs, or outputs used by the custom constraint and objective functions, return the corresponding output argument as `[]`.

For this example, create a mapping that includes only the pressure states for the three tanks. To do so, define the following custom mapping function.

```
function [indx,indu,indy] = myMapping(opspec)
    indx = [getStateIndex(opspec,'scdTanks/Tank1');
            getStateIndex(opspec,'scdTanks/Tank2');
            getStateIndex(opspec,'scdTanks/Tank3')];
    indu = [];
    indy = [];
end
```

Add the custom mapping to the operating point specification.

```
opspec.CustomMappingFcn = @myMapping;
```

When you use a custom mapping function, the indices for the states, inputs, and outputs in your custom constraint and objective functions must be relative to the order specified in the mapping function. Update the custom constraint and objective functions with the new mapping.

```
function [c_ineq,c_eq] =  myConstraintsMap(x,u,y)
    c_ineq = [];
    c_eq = [x(1)-x(2);      % Tank1 pressure - Tank2 pressure
            x(2)-x(3)+1];   % Tank2 pressure - Tank3 pressure + 1
end
```

```
function F =  myObjectiveMap(x,u,y)
    F = max(x(3)-20, 0) + max(16-x(3), 0);
end
```

Here, `x`, `u`, and `y` are vectors of mapped states, inputs, and outputs, respectively. These vectors contain the mapped values specified in `indx`, `indu`, and `indy`, respectively.

Add the updated custom functions to the operating point specification.

```
opspec.CustomConstrFcn = @myConstraintsMap;
opspec.CustomObjFcn = @myObjectiveMap;
```

Trim the model using the custom mapping, and view the trimmed states, which match the previous results in `op2`.

```
[op3,rpt3] = findop(mdl,opspec,opt);
op3.States
```

```
(1.) scdTanks/Inertia
      x: 0
(2.) scdTanks/Tank1
      x: 15
(3.) scdTanks/Tank2
      x: 15
(4.) scdTanks/Tank3
      x: 16
```

Add Analytic Gradients to Custom Functions

For faster or more reliable computations, you can add analytic gradients to your custom constraint and objective functions. Adding gradients can reduce the number of function calls during optimization and potentially improve the accuracy of the optimization result. If you specify gradients, you must specify them for both the custom constraint and objective functions. (Gradients for custom trimming are not supported for Simscape™ models.)

To define the gradient of a given constraint or objective function, take the derivative of the function with respect to a given state, input, or output. For example, if the objective function is

```
F = (u(1)+3)^2 + y(1)^2
```

then the gradient of `F` with respect to `u(1)` is

```
G = 2*(u(1)+3)
```

To add gradients to your custom constraint function, specify the following additional output arguments:

- `G_ineq` - Gradient array for the inequality constraints
- `G_eq` - Gradient array for the equality constraints

Each column of `G_ineq` and `G_eq` contains the gradients for one constraint, and the order of the columns matches the order of the rows in the corresponding constraint vector. The number of rows in both `G_ineq` and `G_eq` is equal to the total number of states, inputs, and outputs in `x`, `u`, and `y`. Each column contains gradients with respect to the states in `x`, followed by the inputs in `u`, then the outputs in `y`.

For this example, add gradients to the constraint function that uses the custom mapping. You do not have to specify a custom mapping when using gradients. However, defining gradients is simpler when using mapped subsets of states, inputs, and outputs.

```
function [c_ineq,c_eq,G_ineq,G_eq] =  myConstraintsGrad(x,u,y)
    c_ineq = [];
    c_eq = [x(1)-x(2);      % Tank1 pressure - Tank2 pressure
            x(2)-x(3)+1];  % Tank2 pressure - Tank3 pressure + 1

    G_ineq = [];
    G_eq = [1  0;
            -1 1;
            0 -1];
end
```

In this function, row `i` of `G_eq` contains gradients with respect to state `x(i)`.

Similarly, to add gradients to your custom objective function, specify an additional output argument `G`, which contains the gradients of `F`. `G` is returned as a column vector with the same format as the columns of `G_ineq` and `G_eq`.

```
function [F,G] = myObjectiveGrad(x,u,y)
    F = max(x(3)-20, 0) + max(16-x(3), 0);

    if x(3) >= 20
        G = [0 0 1]';
```

```
    elseif x(3) <= 16
        G = [0 0 -1]';
    else
        G = [0 0 0]';
    end
end
```

Because the objective function in this example is piecewise differentiable, the value of `G` depends on the value of the pressure in Tank3.

Add the updated custom functions to the operating point specification.

```
opspec.CustomConstrFcn = @myConstraintsGrad;
opspec.CustomObjFcn = @myObjectiveGrad;
```

To enable gradients in the optimization algorithm, turn on the `Jacobian` optimization option.

```
opt.OptimizationOptions.Jacobian = 'on';
```

Trim the model using the custom functions with gradients, and view the trimmed states.

```
[op4,rpt4] = findop(mdl,opspec,opt);
op4.States
```

```
(1.) scdTanks/Inertia
      x: 0
(2.) scdTanks/Tank1
      x: 15
(3.) scdTanks/Tank2
      x: 15
(4.) scdTanks/Tank3
      x: 16
```

The optimization result is the same as the result for the nongradient solution.

To see if the gradients improved the optimization efficiency, view the operating point search reports. For example, compare the number function evaluations for the solution:

- Without gradients:

```
rpt3.OptimizationOutput.funcCount


ans =
```

```
    25
```

- With gradients:

```
rpt4.OptimizationOutput.funcCount


ans =

     5
```

Adding the analytical gradients decreased the number of function calls during optimization.

See Also

`findop` | `getInputIndex` | `getOutputIndex` | `getStateIndex` | `operspec`

More About

- “Compute Steady-State Operating Point from State Specifications” on page 1-14
- “Compute Steady-State Operating Point from Output Specifications” on page 1-28

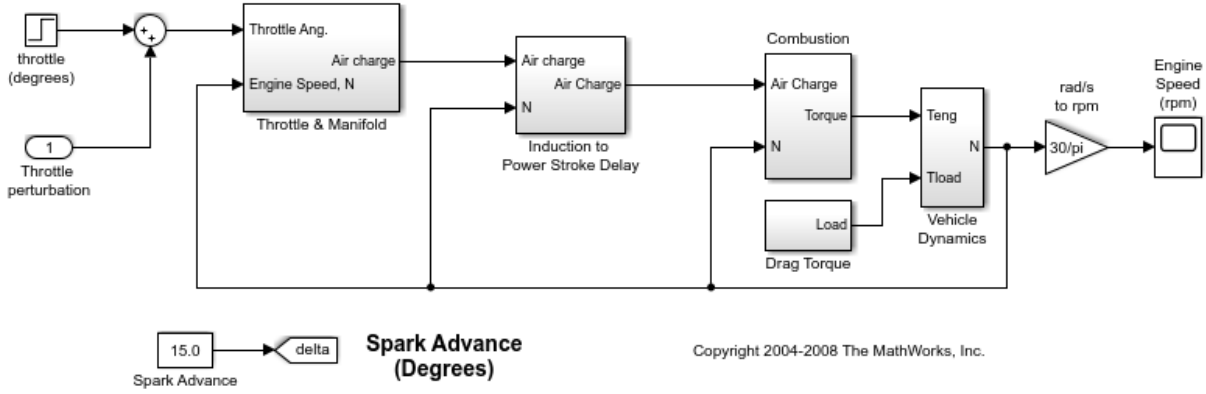
Batch Compute Steady-State Operating Points for Multiple Specifications

This example shows how to find operating points for multiple operating point specifications using the `findop` command. You can batch linearize the model using the operating points and study the change in model behavior.

Each time you call `findop`, the software compiles the Simulink model. To find operating points for multiple specifications, you can give `findop` an array of operating point specifications, instead of repeatedly calling `findop` within a for loop. The software uses a single model compilation to compute the multiple operating points, which is efficient, especially for models that are expensive to recompile repeatedly.

Open the Simulink model.

```
sys = 'scdspeed';
open_system(sys)
```



Create an array of default operating point specification objects.

```
opspec = operspec(sys,3);
```

To find steady-state operating points at which the output of the rad/s to rpm block is fixed, add a known output specification to each operating point specification object.

```
opspec = addoutputspec(opspec,[sys '/rad//s to rpm'],1);
for i = 1:3
```

```
    opspec(i).Outputs(1).Known = true;
end
```

Specify different known output values for each operating point specification.

```
opspec(1).Outputs(1).y = 1500;
opspec(2).Outputs(1).y = 2000;
opspec(3).Outputs(1).y = 2500;
```

Alternatively, you can configure operating point specifications using the Linear Analysis Tool and export the specifications to the MATLAB workspace. For more information, see “Import and Export Specifications For Operating Point Search” on page 1-48.

Find the operating points that meet each of the three output specifications. `findop` computes all the operating points using a single model compilation.

```
ops = findop(sys,opspec);


 Operating point search report:
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.596      dx:      3.41e-09 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           157      dx:     -5.57e-07 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:         -1.61    [-Inf Inf]

Outputs:
----------
(1.) scdspeed/rad//s to rpm
      y:       1.5e+03    (1.5e+03)


 Operating point search report:
```

```
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.544      dx:      2.66e-13 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           209      dx:     -8.48e-12 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:       0.00382    [-Inf Inf]

Outputs:
----------
(1.) scdspeed/rad//s to rpm
      y:         2e+03    (2e+03)


 Operating point search report:
---------------------------------

 Operating point search report for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.511      dx:      1.33e-08 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           262      dx:     -7.83e-08 (0)

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
      u:           1.5    [-Inf Inf]

Outputs:
```

```
----------
(1.) scdspeed/rad//s to rpm
      y:       2.5e+03    (2.5e+03)
```

`ops` is a vector of operating points for the `scdspeed` model that correspond to the specifications in `opspec`. The output value for each operating point matches the known value specified in the corresponding operating point specification.

See Also

`findop` | `operspec`

More About

- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-75
- "Batch Compute Steady-State Operating Points for Parameter Variation" on page 1-65
- "Batch Linearize Model at Multiple Operating Points Using linearize Command" on page 3-28
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41

Batch Compute Steady-State Operating Points for Parameter Variation

In this section...

Block parameters configure a Simulink model in several ways. For example, you can use block parameters to specify various coefficients or controller sample times. You can also use a discrete parameter, like the control input to a Multiport Switch block, to control the data path within a model. Varying the value of a parameter helps you understand its impact on the model behavior. Also, you can vary the parameters of a plant model in a control system to study the robustness of the controller to plant variations.

When trimming a model using `findop`, you can specify a set of parameter values for which to trim the model. The full set of values is called a parameter grid or parameter samples. `findop` computes an operating point for each value combination in the parameter grid. You can vary multiple parameters, thus extending the parameter grid dimension.

Which Parameters Can Be Sampled?

You can vary any model parameter with a value given by a variable in the model workspace, the MATLAB workspace, or a data dictionary. In cases where the varying parameters are all tunable (Simulink), `findop` requires only one model compilation to find operating points for varying parameter values. This efficiency is especially advantageous for models that are expensive to compile repeatedly.

Vary Single Parameter

To vary the value of a single parameter for batch trimming with `findop`, specify the parameter grid as a structure having two fields. The `Name` field contains the name of the

workspace variable that specifies the parameter. The `Value` field contains a vector of values for that parameter to take during trimming.

For example, the `Watertank` model has three parameters defined as MATLAB workspace variables, `a`, `b`, and `A`. The following commands specify a parameter grid for the single parameter for `A`.

```
param.Name = 'A';
param.Value = Avals;
```

Here, `Avals` is an array specifying the sample values for `A`.

The following table lists some common ways of specifying parameter samples.

Parameter Sample-Space Type	How to Specify the Parameter Samples
Linearly varying	`param.Value = linspace(A_min,A_max,num_samples)`
Logarithmically varying	`param.Value = logspace(A_min,A_max,num_samples)`
Random	`param.Value = rand(1,num_samples)`
Custom	`param.Value = custom_vector`

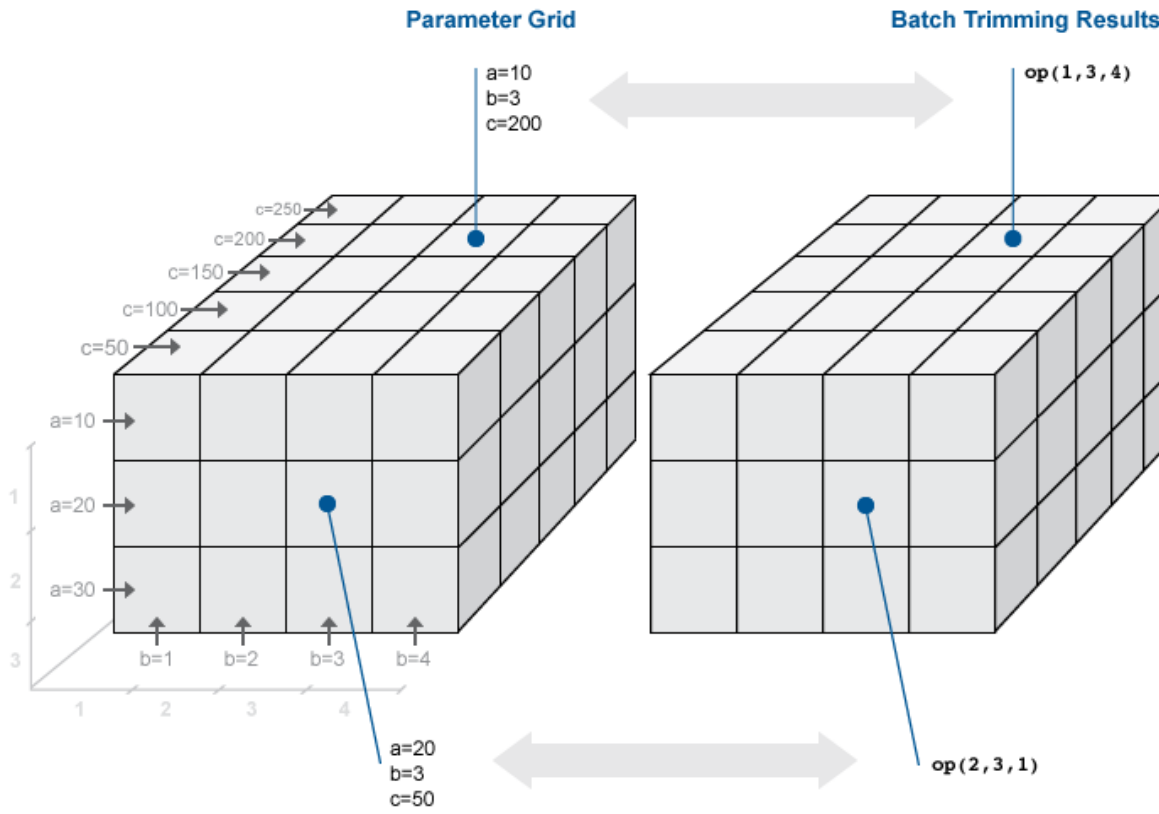
If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, suppose that `Kpid` is a vector of PID gains. The first entry in that vector, `Kpid`, is used as a gain value in a block in your model. Use the following commands to vary that gain using the values given in a vector `Kpvals`:

```
param.Name = 'Kpid(1)';
param.Value = Kpvals;
```

After you create the structure `param`, pass it to `findop` as the `param` input argument.

Multidimension Parameter Grids

When you vary more than one parameter at a time, you generate parameter grids of higher dimension. For example, varying two parameters yields a parameter matrix, and varying three parameters yields a 3-D parameter grid. Consider the following parameter grid used for batch trimming:



Here, you vary the values of three parameters, *a*, *b*, and *c*. The samples form a 3-by-4-by-5 grid. `op` is an array with same dimensions that contains corresponding trimmed operating point objects.

Vary Multiple Parameters

To vary the value of multiple parameters for batch trimming with `findop`, specify parameter samples as a structure array. The structure has an entry for each parameter whose value you vary. The structure for each parameter is the same as described in "Vary Single Parameter" on page 1-65. You can specify the `Value` field for a parameter

as an array of any dimension. However, the size of the `Value` field must match for all parameters. Corresponding array entries for all the parameters, also referred to as a parameter grid points, must map to a specified parameter combination. When the software trims the model, it computes an operating point for each grid point.

Specify Full Grid

Suppose that your model has two parameters whose values you want to vary, a and b:

$a = \{a1, a2\}$

$b = \{b1, b2\}$

You want to trim the model for every combination of a and b, also referred to as a full grid:

$$\begin{Bmatrix} (a_1, b_1), & (a_1, b_2) \\ (a_2, b_1), & (a_2, b_2) \end{Bmatrix}$$

Create a rectangular parameter grid using `ndgrid`.

```
a1 = 1;
a2 = 2;
a = [a1 a2];

b1 = 3;
b2 = 4;
b = [b1 b2];

[A,B] = ndgrid(a,b)

>> A

A =

     1     1
     2     2

>> B

B =

     3     4
     3     4
```

Create the structure array, `params`, that specifies the parameter grid.

```
params(1).Name = 'a';
params(1).Value = A;

params(2).Name = 'b';
params(2).Value = B;
```

In general, to specify a full grid for *N* parameters, use `ndgrid` to obtain *N* grid arrays.

```
[P1,...,PN] = ndgrid(p1,...,pN);
```

Here, p_1,...,p_N are the parameter sample vectors.

Create a 1 x *N* structure array.

```
params(1).Name = 'p1';
params(1).Value = P1;
...
params(N).Name = 'pN';
params(N).Value = PN;
```

Specify Subset of Full Grid

If your model is complex or you vary the value of many parameters, trimming the model for the full grid can become expensive. In this case, you can specify a subset of the full grid using a table-like approach. Using the example in "Specify Full Grid" on page 1-68, suppose that you want to trim the model for the following combinations of *a* and *b*:
$\{(a_1,b_1),(a_1,b_2)\}$

Create the structure array, `params`, that specifies this parameter grid.

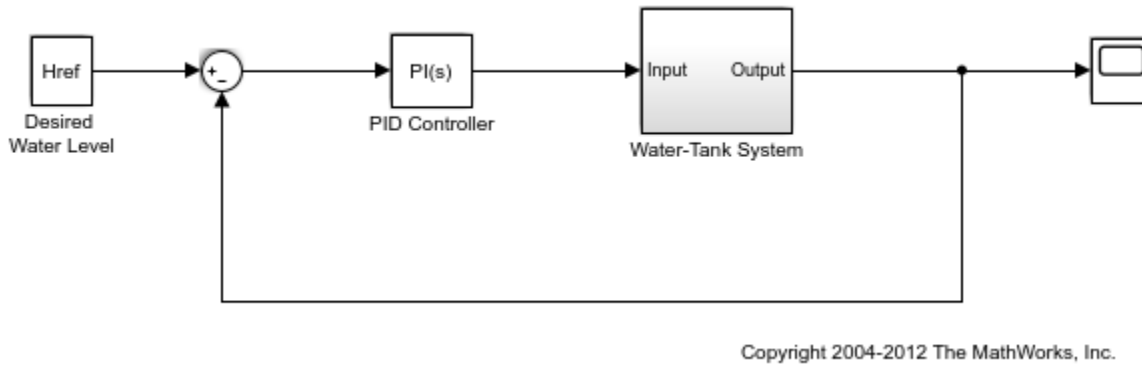
```
A = [a1 a1];
params(1).Name = 'a';
params(1).Value = A;

B = [b1 b2];
params(2).Name = 'b';
params(2).Value = B;
```

Batch Trim Model for Parameter Variations

This example shows how to obtain multiple operating points for a model by varying parameter values. You can study the controller robustness to plant variations by batch linearizing the model using the trimmed operating points.

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Vary parameters `A` and `b` within 10% of their nominal values. Specify three values for `A` and four values for `b`, creating a 3-by-4 value grid for each parameter.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the model, which specifies that both model states are unknown and must be at steady state in the trimmed operating point.

```
opspec = operspec(sys)
```

```
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) watertank/PID Controller/Integrator
     spec:  dx = 0,  initial guess: 0
(2.) watertank/Water-Tank System/H
     spec:  dx = 0,  initial guess: 1

Inputs: None
----------

Outputs: None
----------
```

By default, `findop` displays an operating point search report in the Command Window for each trimming operation. To suppress the report display, create a trimming option set and turn off the operating point search report display.

```
opt = findopOptions('DisplayReport','off');
```

Trim the model using the specified operating point specification, parameter grid, and option set.

```
[op,opreport] = findop(sys,opspec,params,opt);
```

`findop` trims the model for each parameter combination. The software uses only one model compilation. `op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

View the operating point in the first row and first column of `op`.

```
op(1,1)
```

```
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
```

```
(1.) watertank/PID Controller/Integrator
      x: 1.41
(2.) watertank/Water-Tank System/H
      x: 10

Inputs: None
----------
```

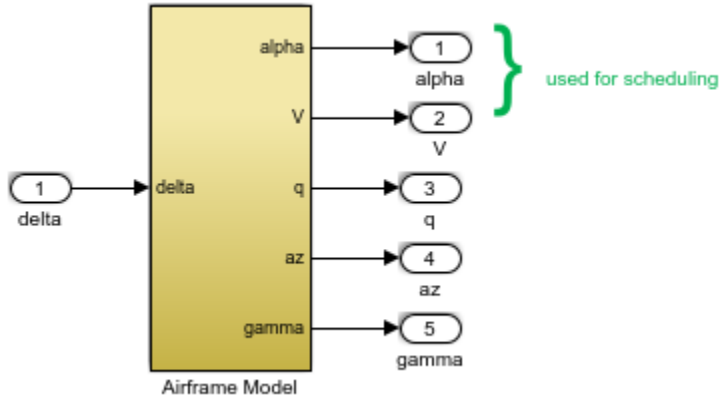
Batch Trim Model at Known States Derived from Parameter Values

This example shows how to batch trim a model when the specified parameter variations affect the known states for trimming.

In the "Batch Trim Model for Parameter Variations" on page 1-70 example, the model is trimmed to meet a single operating point specification that contains unknown states. In other cases, the model states are known for trimming, but depend on the values of the varying parameters. In this case, you cannot batch trim the model using a single operating point specification. You must create a separate specification for each parameter value grid point.

Open the Simulink model.

```
sys = 'scdairframeTRIM';
open_system(sys)
```



In this model, the aerodynamic forces and moments depend on the speed, V, and incidence, α.

Vary the V and α parameters, and create a 6-by-4 parameter grid.

```
nA = 6;     % number of alpha values
nV = 4;     % number of V values
alphaRange = linspace(-20,20,nA)*pi/180;
vRange = linspace(700,1400,nV);
[alphaGrid,vGrid] = ndgrid(alphaRange,vRange);
```

Since some known state values for trimming depend on the values of V and α, you must create a separate operating point specification object for each parameter combination.

```
for i = 1:nA
    for j = 1:nV
        % Set parameter values in model.
        alpha_ini = alphaGrid(i,j);
        v_ini = vGrid(i,j);

        % Create default specifications based on the specified parameters.
        opspec(i,j) = operspec(sys);

        % Specify which states are known and which states are at steady state.
        opspec(i,j).States(1).Known = [1;1];
        opspec(i,j).States(1).SteadyState = [0;0];

        opspec(i,j).States(3).Known = [1;1];
        opspec(i,j).States(3).SteadyState = [0;1];

        opspec(i,j).States(2).Known = 1;
        opspec(i,j).States(2).SteadyState = 0;

        opspec(i,j).States(4).Known = 0;
        opspec(i,j).States(4).SteadyState = 1;
    end
end
```

Create a parameter structure for batch trimming. Specify a name and value grid for each parameter.

```
params(1).Name = 'alpha_ini';
params(1).Value = alphaGrid;
params(2).Name = 'v_ini';
params(2).Value = vGrid;
```

Trim the model using the specified parameter grid and operating point specifications. When you specify an array of operating point specifications and varying parameter

values, the dimensions of the specification array must match the parameter grid dimensions.

```
opt = findopOptions('DisplayReport','off');
op = findop(sys,opspec,params,opt);
```

`findop` trims the model for each parameter combination. `op` is a 6-by-4 array of operating point objects that correspond to the specified parameter grid points.

See Also

`findop` | `linearize` | `operspec`

More About

- "Batch Compute Steady-State Operating Points for Multiple Specifications" on page 1-61
- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-75
- "Batch Linearize Model at Multiple Operating Points Using linearize Command" on page 3-28
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41

Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code

This example shows how to batch compute steady-state operating points for a model using generated MATLAB code. You can batch linearize a model using the operating points and study the change in model behavior.

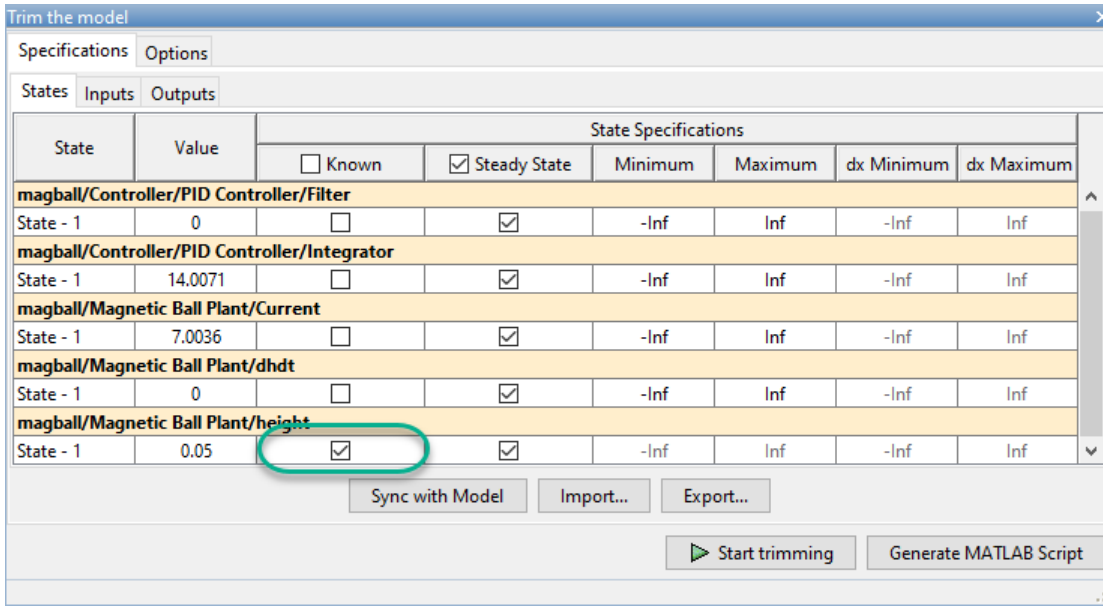
If you are new to writing scripts, interactively configure your operating points search using the Linear Analysis Tool. You can use Simulink Control Design to automatically generate a script based on your Linear Analysis Tool settings.

1. Open the Simulink model.

   ```
   sys = 'magball';
   open_system(sys)
   ```

2. In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.
3. In the Linear Analysis Tool, in the **Operating Point** drop-down list, click `Trim Model`.

   By default, the software specifies all model states to be at equilibrium, as shown in the **Steady State** column.
4. In the **States** tab, select the **Known** check box for the **magball/Magnetic Ball Plant/height** state.



5 Click **Start trimming** to compute the operating point using numerical optimization.

The Trim progress viewer shows that the optimization algorithm terminated successfully. The `(Maximum Error) Block` area shows the progress of reducing the error of a specific state or output during the optimization.

6 In the Trim the model dialog box, click **Generate MATLAB Script**

The MATLAB Editor window opens with an automatically generated script.

7 Modify the script to trim the model at multiple operating points.

- **a** Remove unneeded comments from the generated script.
- **b** Define the height variable, `height`, with values at which to compute operating points.
- **c** Add a `for` loop around the operating point search code to compute a steady-state operating point for each `height` value. Within the loop, before calling `findop`, update the reference ball height, specified by the Desired Height block.

Your script should now look similar to this:

```
%% Specify the model name
model = 'magball';
```

```
%% Create the operating point specification object.
opspec = operspec(model);

% State (5) - magball/Magnetic Ball Plant/height
% - Default model initial conditions are used to initialize optimization.
opspec.States(5).Known = true;

%% Create the options
opt = findopOptions('DisplayReport','iter');

%% Specify ball heights at which to compute operating points
height = [0.05;0.1;0.15];

%% Loop over height values to find the corresponding operating points
for i = 1:length(height)
    % Set the ball height in the specification
    opspec.States(5).x = height(i);

    % Update the model ball haight reference parameter
    set_param('magball/Desired Height','Value',num2str(height(i)))

    % Trim the model
    [op(i),opreport(i)] = findop(model,opspec,opt);
end
```

After running this script, `op` contains operating points corresponding to each of the specified `height` values.

See Also

`findop`

More About

- "Generate MATLAB Code for Operating Point Configuration" on page 1-101
- "Batch Linearize Model at Multiple Operating Points Using linearize Command" on page 3-28
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41
- "Batch Compute Steady-State Operating Points for Multiple Specifications" on page 1-61
- "Batch Compute Steady-State Operating Points for Parameter Variation" on page 1-65

Compute Operating Points at Simulation Snapshots

In this section...

You can compute a steady-state operating point using a model simulation. The resulting operating point consists of the state values and model input levels at a specified simulation snapshot time.

To use simulation-based operating point computation, first configure your model initial conditions such that the model converges to equilibrium. You can then simulate your model and create operating points interactively using the Linear Analysis Tool or programmatically at the MATLAB command line.

To verify that the operating point is at steady state, initialize your model with the operating point values, simulate the model, and check if key signals and states are at equilibrium. For more information on initializing your model with an operating point, see “Simulate Simulink Model at Specific Operating Point” on page 1-83.

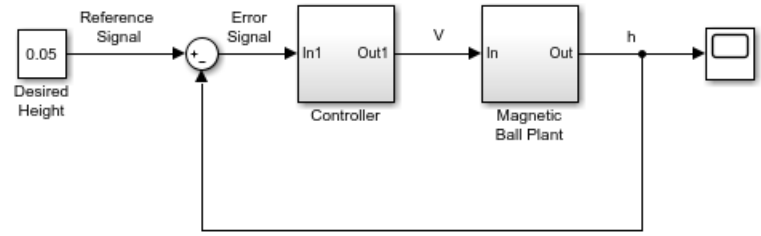
Note If your Simulink model contains blocks with internal states, do not linearize the model at an operating point you compute from a simulation snapshot. Instead, try linearizing the model using a simulation snapshot or at an operating point found using trimming.

Compute Operating Points at Simulation Snapshots Using Linear Analysis Tool

This example shows how to compute an operating point at specified simulation snapshot times using the Linear Analysis Tool.

Open the Simulink model.

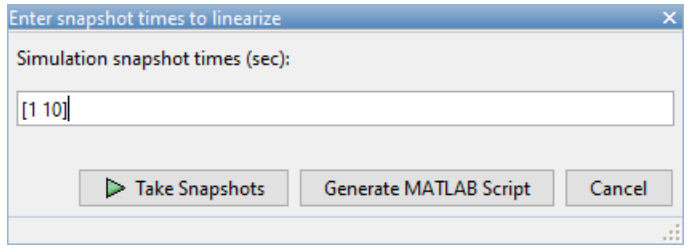
```
sys = 'magball';
open_system(sys)
```



To open the Linear Analysis Tool, in the Simulink model window, select **Analysis** > **Control Design** > **Linear Analysis**.

To specify the simulation snapshot time, in the Linear Analysis Tool, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Take Simulation Snapshot`.

Take simulation snapshots at `1` and `10` time units. In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** box, enter `[1,10]`.

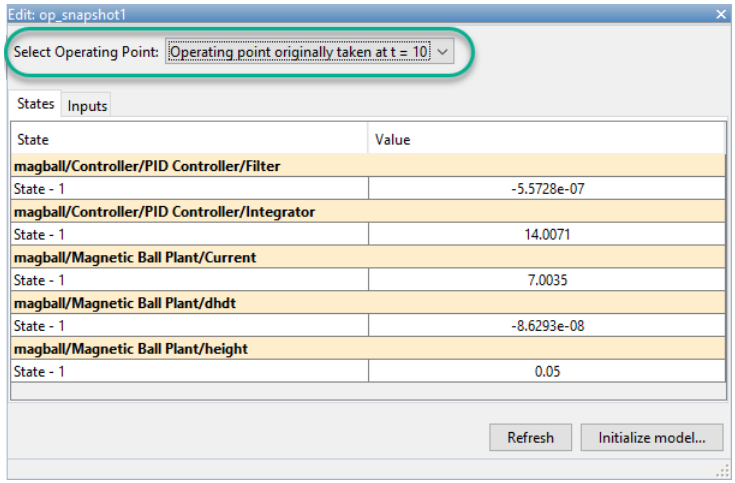


To take the snapshots, click **Take Snapshots**.

The software simulates the model and creates an operating point at each simulation snapshot time. Each operating point contains the input and states values of the model at the corresponding snapshot time.

An array of operating points, `op_snapshot1`, appears in the **Linear Analysis Workspace**. This array contains two operating points, one for each specified snapshot time.

To view the operating points, in the **Linear Analysis Workspace**, double-click `op_snapshot1`. You can select which operating point to view using the **Select Operating Point** drop-down list.

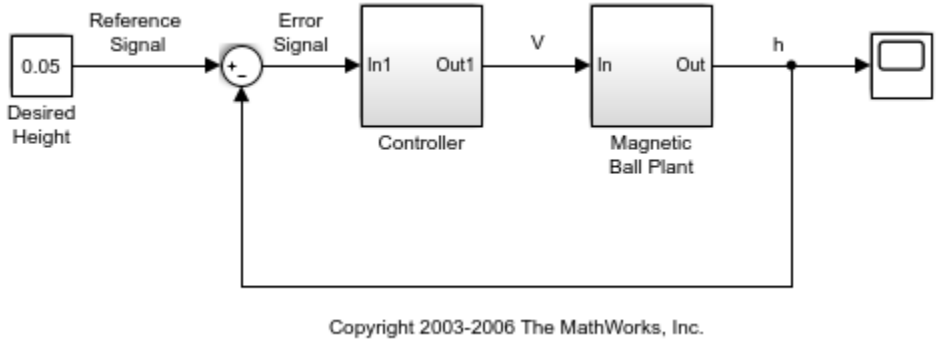


Find Operating Points at Simulation Snapshots at Command Line

This example shows how to compute a steady-state operating point at specified simulation snapshot times.

Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```



Simulate the model, and create operating points at `1` and `10` time units. The software simulates the model and computes an operating point at each simulation snapshot time.

```
op = findop(sys,[1 10]);
```

`op` is a column vector of operating points, with one element for each specified snapshot time.

Display the first operating point.

```
op(1)
```

```
 Operating point for the Model magball.
 (Time-Varying Components Evaluated at time t=1)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 5.76e-06
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: -6.7e-08
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

See Also

Apps
Linear Analysis Tool

Functions
`findop`

More About

- “About Operating Points” on page 1-2
- “Simulate Simulink Model at Specific Operating Point” on page 1-83

- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-41

Simulate Simulink Model at Specific Operating Point

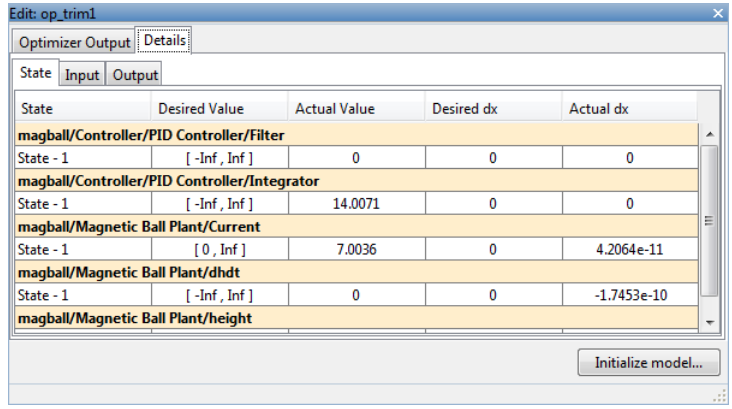
This example shows how to initialize a model at a specific operating point for simulation.

1 Compute a steady-state operating point using one of the following:

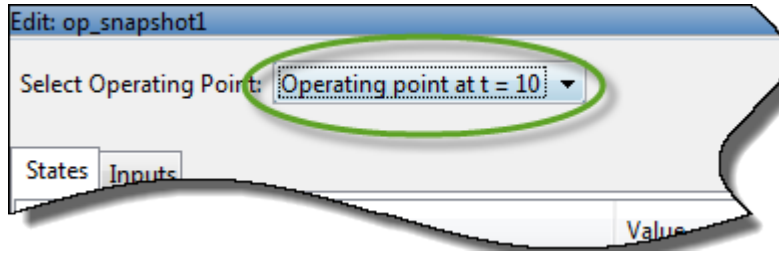
- State specifications, see “Compute Steady-State Operating Point from State Specifications” on page 1-14
- Output Specifications, see “Compute Steady-State Operating Point from Output Specifications” on page 1-28
- Simulation snapshot, see “Compute Operating Points at Simulation Snapshots” on page 1-78.

2 In the Linear Analysis Tool, double-click the computed operating point or simulation snapshot variable in the **Linear Analysis Workspace**.

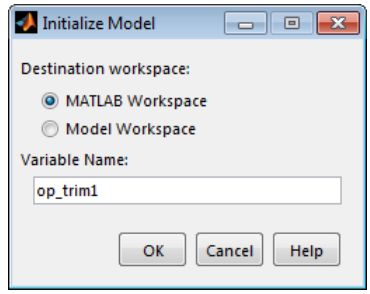
The Edit dialog box opens.



Note If you computed multiple operating points using a simulation snapshot. Select an operating point from the **Select Operating Point** list.



3 Click **Initialize model**.



In the Initialize Model dialog box, specify a **Variable Name** for the operating point object. Alternatively, you can use the default variable name.

Click **OK** to export the operating point to the MATLAB Workspace.

This action also sets the operating point values in the **Data Import/Export** pane of the Configuration Parameters dialog box. Simulink derives the initial conditions from this operating point when simulating the model.

Tip If you want to store this operating point with the model, export the operating point to the **Model Workspace** instead.

In the Simulink Editor, select **Simulation > Run** to simulate the model starting at the specified operating point.

See Also

Related Examples

- "Compute Steady-State Operating Point from State Specifications" on page 1-14
- "Compute Steady-State Operating Point from Output Specifications" on page 1-28
- "Compute Operating Points at Simulation Snapshots" on page 1-78

Handle Blocks with Internal State Representation

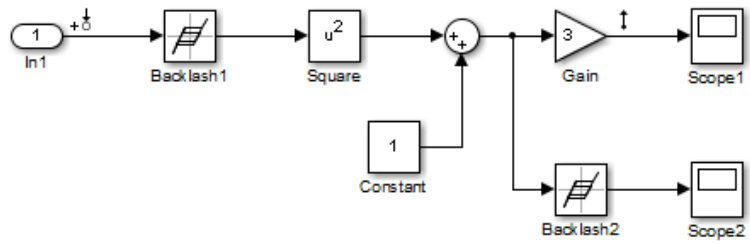
In this section...

Operating Point Object Excludes Blocks with Internal States

The operating point object used for linearization and control design does not include these Simulink blocks with internal state representation:

- Memory blocks
- Transport Delay and Variable Transport Delay blocks
- Disabled Action Subsystem (Simulink) blocks
- Backlash blocks
- MATLAB Function blocks with persistent data
- Rate Transition blocks
- Stateflow blocks
- S-Function blocks with states not registered as Continuous or Double Value Discrete

For example, if you compute a steady-state operating point for the following Simulink model, the resulting operating point object *does not* include the Backlash block states because these states have an internal representation. If you use this operating point object to initialize a Simulink model, the initial conditions of the Backlash blocks might be incompatible with the operating point.



Identifying Blocks with Internal States in Your Model

Generate a list of blocks that have internal state representations.

```
sldiagnostics(sys,'CountBlocks')
```

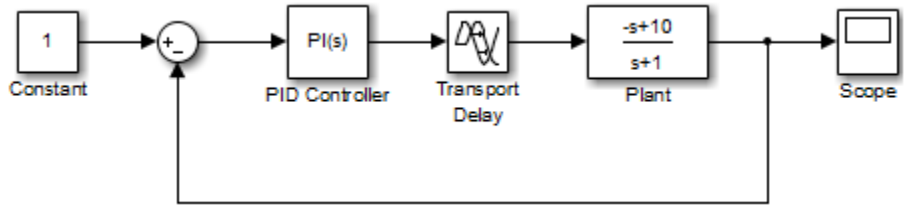
where `sys` is the model name. This command also returns the number of occurrences of each block.

Configuring Blocks with Internal States for Steady-State Operating Point Search

Blocks with internal states can cause problems for steady-state operating point search (trimming). Where there is *no direct feedthrough*, the input to the block at the current time does not determine the output of the block at the current time.

To fix this issue for Memory, Transport Delay, or Variable Transport Delay blocks, select the **Direct feedthrough of input during linearization** option in the Block Parameters dialog box before searching for an operating point or linearizing a model at a steady state. This setting makes such blocks behave as if they have a gain of one during an operating point search.

For example, the next model includes a Transport Delay block. In this case, you cannot find a steady state operating point using optimization because the output of the Transport Delay is always zero. Because the reference signal is 1, the input to the Plant block must be nonzero to get the plant block to have an output of 1 and be at steady state.



Select the **Direct feedthrough of input during linearization** option in the Block Parameters dialog box before searching for an operating point. This setting allows the PID Controller block to pass a nonzero value to the Plant block.

You can also set direct feedthrough options at the command-line.

Block	Command to specify direct feedthrough
Memory	`set_param(blockname,'LinearizeMemory','on')`
Transport Delay or Variable Transport Delay	`set_param(blockname,'TransDelayFeedthrough','on')`

For other blocks with internal states, determine whether the output of the block impacts the state derivatives or desired output levels before computing operating points. If the block impacts these derivatives or output levels, consider replacing it using a configurable subsystem.

See Also

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Points” on page 1-6

Synchronize Simulink Model Changes with Operating Point Specifications

Modifying your Simulink model can change, add, or remove states, inputs, or outputs, which changes the operating point. You can synchronize existing operating point specification objects to reflect the changes in your model.

In this section...

Synchronize Simulink Model Changes Using Linear Analysis Tool

If you change your Simulink model while the Linear Analysis Tool is open, you must sync the operating point specifications in the Linear Analysis Tool to reflect the changes in the model.

Modifying your Simulink model can change, add, or remove states, inputs, or outputs, which changes the operating point.

1 Open the Simulink model.

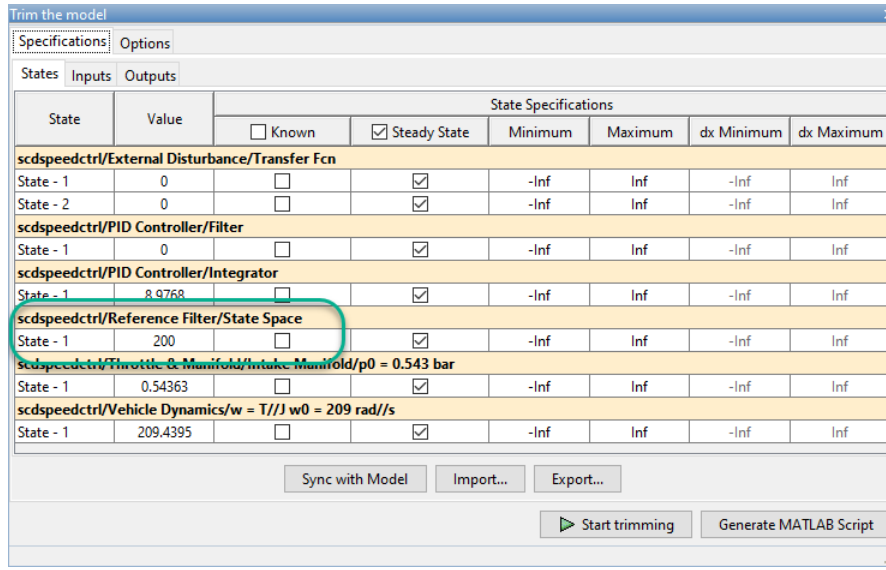
```
sys = ('scdspeedctrl');
open_system(sys)
```

2 In the Simulink Editor, select **Analysis** > **Control Design** > **Linear Analysis**.

The Linear Analysis Tool for the model opens, with the default operating point being set to the model initial condition.

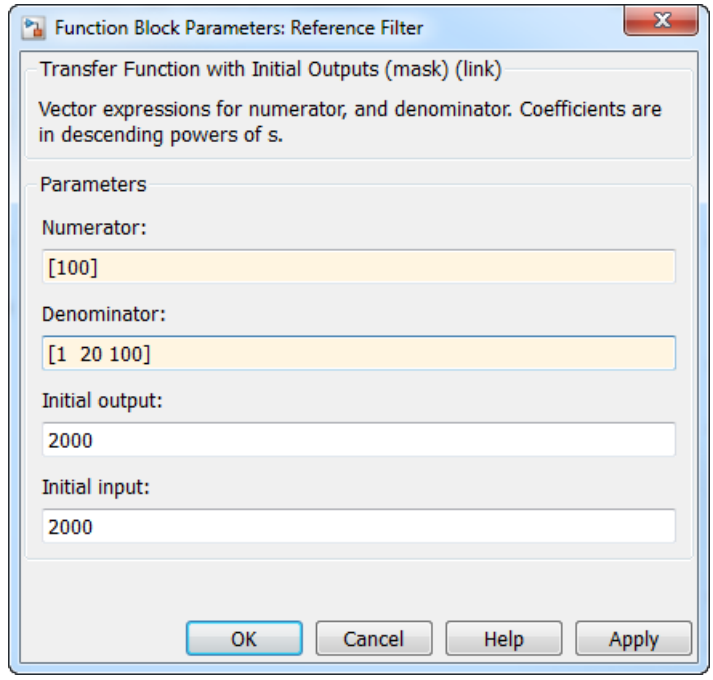
3 In the Linear Analysis Tool, in the **Operating Points** drop-down list, select **Trim Model**.

The Trim the model dialog box appears.



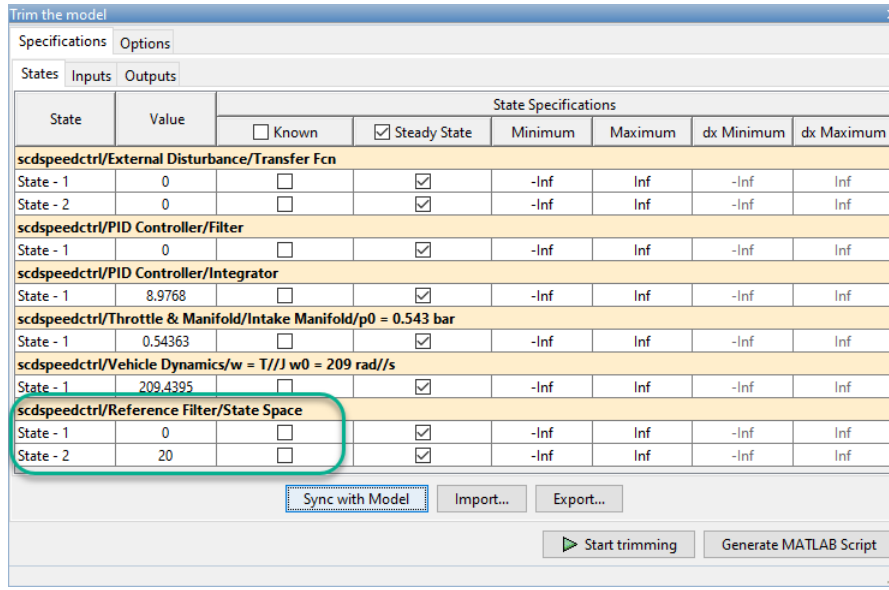
The Reference Filter block contains just one state.

4 In the Simulink Editor, double-click the Reference Filter block. Change the **Numerator** of the transfer function to `[100]`, and change the **Denominator** to `[1 20 100]`. Click **OK**.



This change increases the order of the filter, adding a state to the Simulink model.

5 In the Trim the model dialog, click **Sync with Model** to synchronize the operating point specifications in the Linear Analysis Tool with the updated model states.



The dialog now shows two states for the Reference Filter block.

6 To compute the operating point, click **Start trimming**.

Synchronize Simulink Model Changes at the Command Line

This example shows how to update an existing operating point specification object with changes in the Simulink model.

1 Open the Simulink model.

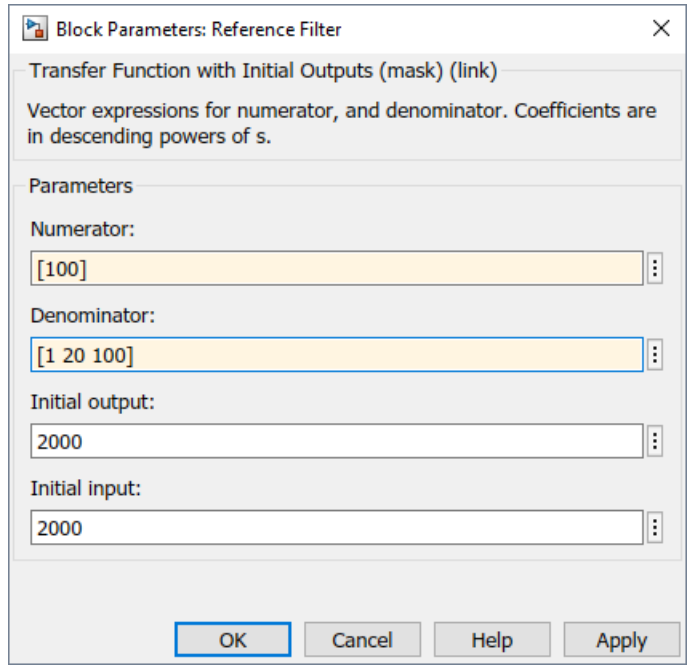
```
sys = 'scdspeedctrl';
open_system(sys)
```

2 Create operating point specification object.

```
opspec = operspec(sys);
```

By default, all model states are specified to be at steady state.

3 In the Simulink Editor, double-click the Reference Filter block. Change the **Numerator** of the transfer function to `[100]` and the **Denominator** to `[1 20 100]`. Click **OK**.



4 Attempt to find the steady-state operating point that meets these specifications.

```
op = findop(sys,opspec);
```

This command results in an error because the changes to your model are not reflected in your operating point specification object:

```
??? The model scdspeedctrl has been modified and the operating point
object is out of date.  Update the object by calling the function
update on your operating point object.
```

5 Update the operating point specification object with changes to the model. Repeat the operating point search.

```
opspec = update(opspec);
op = findop(sys,opspec);
bdclose(sys)

 Operating Point Search Report:
---------------------------------

 Operating Report for the Model scdspeedctrl.
```

```
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) scdspeedctrl/External Disturbance/Transfer Fcn
      x:             0      dx:             0 (0)
      x:             0      dx:             0 (0)
(2.) scdspeedctrl/PID Controller/Filter
      x:             0      dx:            -0 (0)
(3.) scdspeedctrl/PID Controller/Integrator
      x:          8.98      dx:     -4.51e-14 (0)
(4.) scdspeedctrl/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
      x:         0.544      dx:      2.94e-15 (0)
(5.) scdspeedctrl/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           209      dx:     -1.52e-13 (0)
(6.) scdspeedctrl/Reference Filter/State Space
      x:           200      dx:             0 (0)

Inputs: None
----------

Outputs: None
----------
```

After updating the operating point specifications object, the optimization algorithm successfully finds the operating point.

See Also

`update`

More About

- “Simulate Simulink Model at Specific Operating Point” on page 1-83

Find Steady-State Operating Points for Simscape Models

You can find operating points for models with Simscape components using Simulink Control Designsoftware. In particular, you can find steady-state operating points using one of the following methods:

- **Optimization-based trimming** — Specify constraints on model inputs, outputs, or states, and compute a steady-state operating point that satisfies these constraints. For more information, see “Compute Steady-State Operating Point from State Specifications” on page 1-14 and “Compute Steady-State Operating Point from Output Specifications” on page 1-28.

  By default, you can define operating point specifications for any Simulink and Simscape states in your model, and any root-level input and output ports of your model. You can also define additional output specifications on Simulink signals. To apply output specifications to a Simscape physical signal, first convert the signal using a PS-Simulink Converter block.
- **Simulation snapshot** — Specify model initial conditions near an expected equilibrium point, and simulate the model until it reaches steady state. You can then create an operating point based on the steady-state signals and states in the model. For more information, see “Compute Operating Points at Simulation Snapshots” on page 1-78.

Projection-Based Trim Optimizers

To produce better trimming results for Simscape models, you can use projection-based trim optimizers. These optimizers enforce the consistency of the model initial condition at each evaluation of the objective function or nonlinear constraint function. Using projection-based trim optimizers requires Optimization Toolbox™ software.

You can use these projection-based optimizers when trimming models from the command line and in the Linear Analysis Tool.

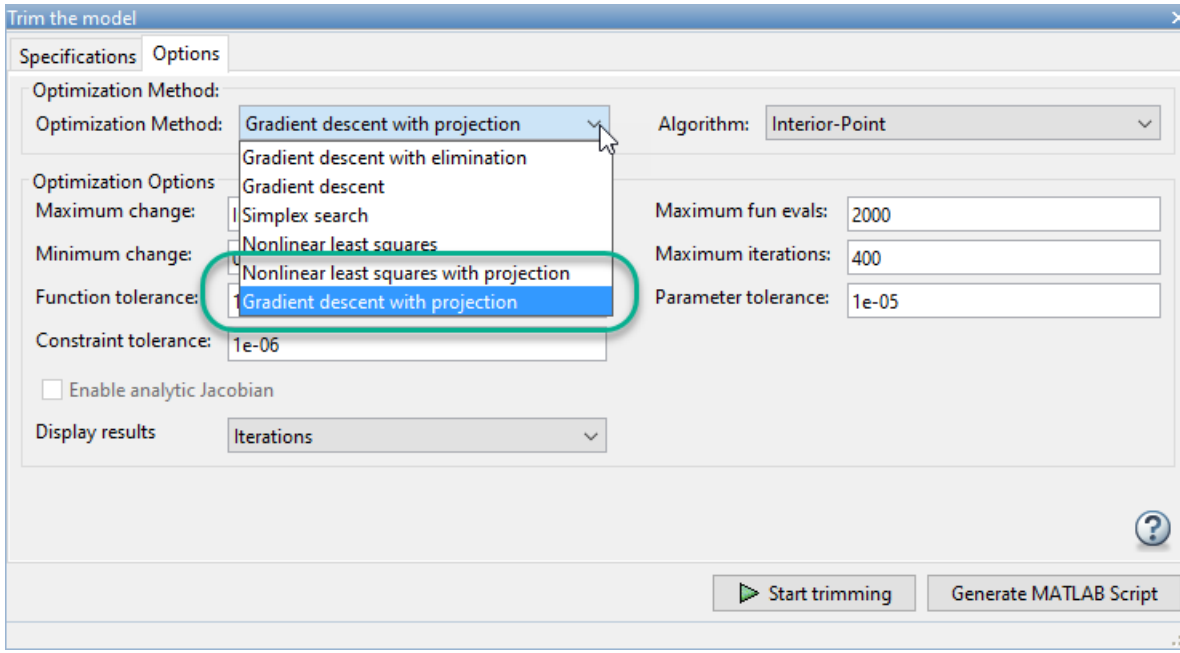
To specify the optimizer type at the command line, create a `findopOptions` option set, and specify the `Optimizer` option as one of the following:

- `'lsqnonlin-proj'` — Nonlinear least squares with projection
- `'graddescent-proj'` — Gradient descent with projection

When using gradient descent with projection at the command line, you can specify whether the algorithm enforces the model initial conditions using hard or soft constraints by specifying the `ConstraintType` option in `findopOptions`.

To specify the optimizer type in the Linear Analysis Tool, first open the Trim the model dialog box. In the Linear Analysis Tool, in the **Operating Point** drop-down list, select `Trim Model`.

Then, in the Trim the model dialog box, on the **Options** tab, in the **Optimization Method** drop-down list, select an optimizer.



When you use gradient descent with projection in the Linear Analysis Tool, the algorithm enforces the model initial conditions using hard constraints.

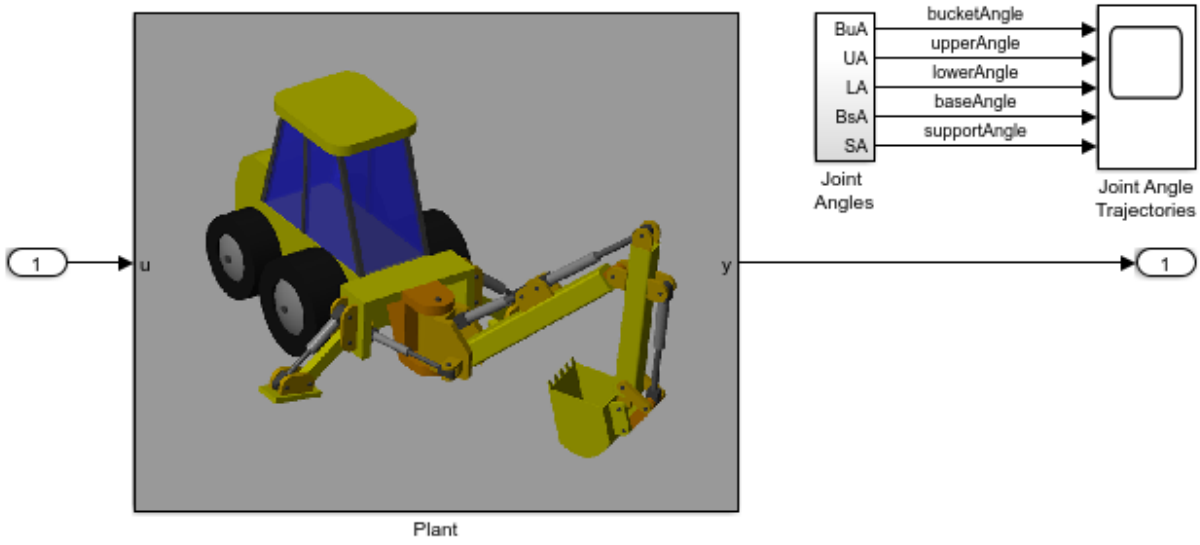
Steady-State Simulation with Projection-Based Trim Optimizer

This example shows how to find a steady-state operating point for a Simscape™ Multibody™ model using `findop` with a projection-based optimizer. Results are verified using simulation.

Open Model

Open the Simulink model.

```
mdl = 'scdbackhoeTRIM';
open_system(mdl)
```



Define Operating Point Specifications

Before creating an operating point specification, configure the model to use the model initial condition.

```
set_param(mdl,'LoadExternalInput','off')
set_param(mdl,'LoadInitialState','off')
```

Create a default operating point specification object.

```
ops = operspec(mdl);
```

Impose constraints on the outputs.

```
ops.Outputs(1).Known = true(10,1);
ops.Outputs(1).y(1) = 0;     % Bucket angle
ops.Outputs(1).y(3) = 50;    % Upper angle
ops.Outputs(1).y(5) = -50;   % Lower angle
ops.Outputs(1).y(7) = 0;     % Base angle
ops.Outputs(1).y(9) = -45;   % Support angle
```

Configure Trim Options

Configure trim optimizer options. Set the `'OptimizerType'` option to `'graddescent-proj'`, which is a projection-based trim optimizer that enforces consistency of the model physical states. To display trim progress, set the `'DisplayReport'` option to `'iter'`.

```
opt = findopOptions('OptimizerType','graddescent-proj',...
                    'DisplayReport','iter');
opt.OptimizationOptions.MaxFunEvals = 20000;
```

Trim Model

Find the steady-state operating point that meets these specifications. The following command takes a few minutes.

```
[op,rpt] = findop(mdl,ops,opt);
```

To save time, load precomputed results.

```
load('scdbackhoe_op.mat')
```

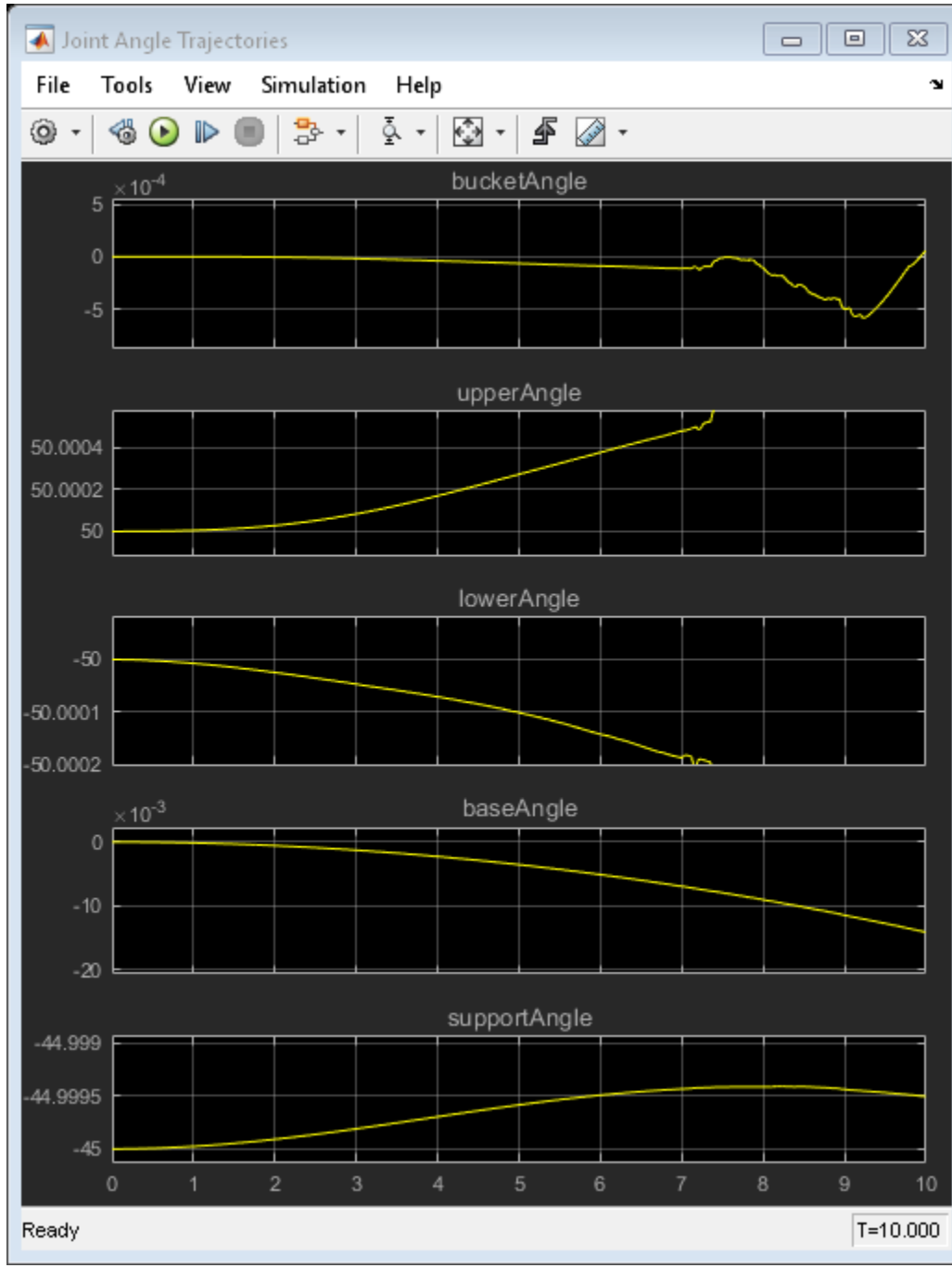
Simulate Model

Simulate the model from the computed steady state.

```
set_param(mdl, 'LoadExternalInput','on')
set_param(mdl, 'ExternalInput','getinputstruct(op)')
set_param(mdl, 'LoadInitialState','on')
set_param(mdl, 'InitialState','getstatestruct(op)')
sim(mdl)
```

Open scope to inspect results.

```
open_system([mdl, '/Joint Angle Trajectories'])
```


Joint Angle Trajectories
File
Tools
View
Simulation
Help
bucketAngle
×10^-4
5
0
-5
upperAngle
50.0004
50.0002
50
lowerAngle
-50
-50.0001
-50.0002
baseAngle
×10^-3
0
-10
-20
supportAngle
-44.999
-44.9995
-45
0
1
2
3
4
5
6
7
8
9
10
Ready
T=10.000

The simulation results show that the five angles are trimmed to their expected values, however the trajectory deviates slightly over time due to numerical noise and instability. You can stabilize the angles using feedback controllers.

```
bdclose(mdl)
```

See Also

Apps

Linear Analysis Tool

Functions

`findop` | `findopOptions` | `operspec`

Blocks

PS-Simulink Converter

More About

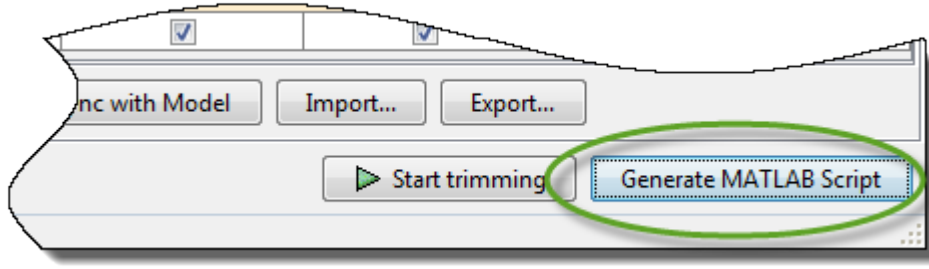
- “About Operating Points” on page 1-2

Generate MATLAB Code for Operating Point Configuration

This topic shows how to generate MATLAB code from the Linear Analysis Tool for operating point configuration. You can generate a MATLAB script to programmatically reproduce a result that you obtained interactively. You can also modify the script to compute multiple operating points with systematic variations in operating point specifications (batch computing).

To generate MATLAB code for configuring operating points:

1. In the Linear Analysis Tool, in the **Linear Analysis** tab, in the **Operating Points** drop-down list, click `Trim Model`.
2. In the Trim the model dialog box, in the **Specifications** tab, configure the operating point state, input, and output specifications.
3. In the **Options** tab, specify search optimization settings.
4. Click **Generate MATLAB Script** to generate code that creates an operating point using your specifications and search options.



You can examine the generated code in the MATLAB Editor. To modify the script to perform batch operating point computation, see “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-75.

See Also

`findop`

More About

- “Compute Steady-State Operating Points” on page 1-6

- “Compute Steady-State Operating Point from State Specifications” on page 1-14
- “Compute Steady-State Operating Point from Output Specifications” on page 1-28
- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-75

Linearization

Linearize Nonlinear Models

In this section...

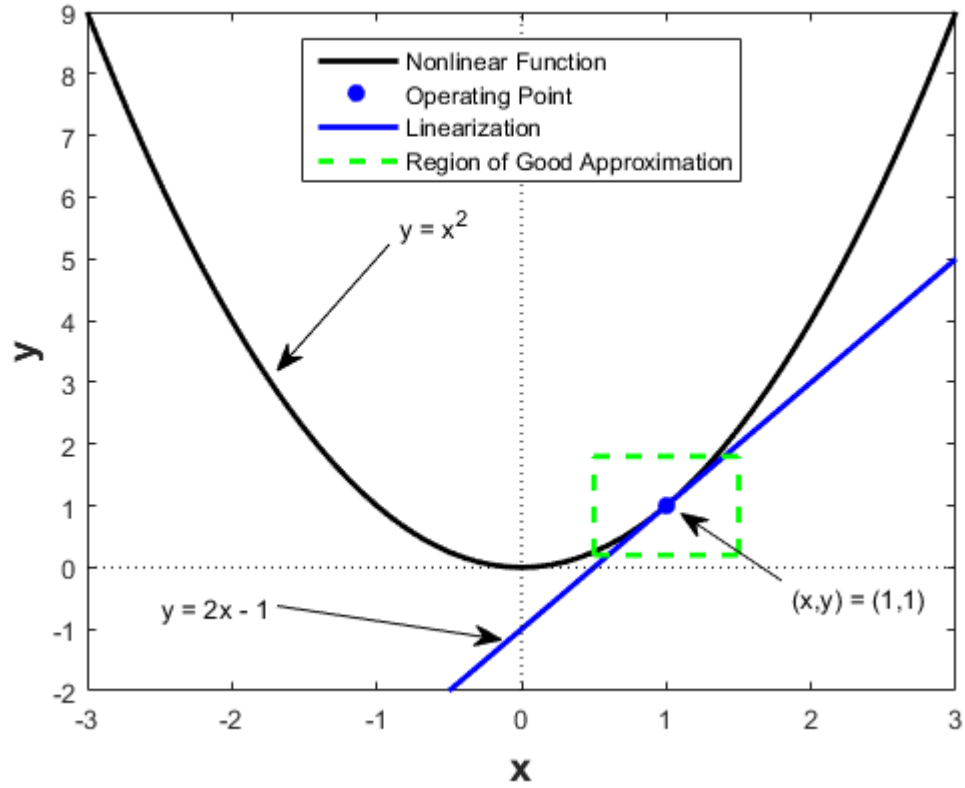
What Is Linearization?

Linearization is a linear approximation of a nonlinear system that is valid in a small region around an operating point.

For example, suppose that the nonlinear function is $y = x^2$. Linearizing this nonlinear function about the operating point $x = 1$, $y = 1$ results in a linear function $y = 2x - 1$.

Near the operating point, $y = 2x - 1$ is a good approximation to $y = x^2$. Away from the operating point, the approximation is poor.

The next figure shows a possible region of good approximation for the linearization of $y = x^2$. The actual region of validity depends on the nonlinear model.



Extending the concept of linearization to dynamic systems, you can write continuous-time nonlinear differential equations in this form:

$$\dot{x}(t) = f\left(x(t),u(t),t\right)$$

$$y(t) = g\left(x(t),u(t),t\right).$$

In these equations, $x(t)$ represents the system states, $u(t)$ represents the inputs to the system, and $y(t)$ represents the outputs of the system.

A linearized model of this system is valid in a small region around the operating point $t=t_0$, $x(t_0)=x_0$, $u(t_0)=u_0$, and $y(t_0)=g(x_0,u_0,t_0)=y_0$.

To represent the linearized model, define new variables centered about the operating point:

$$\delta x(t) = x(t) - x_0$$
$$\delta u(t) = u(t) - u_0$$
$$\delta y(t) = y(t) - y_0$$

The linearized model in terms of δx, δu, and δy is valid when the values of these variables are small:

$$\delta \dot{x}(t) = A\delta x(t) + B\delta u(t)$$
$$\delta y(t) = C\delta x(t) + D\delta u(t)$$

Applications of Linearization

Linearization is useful in model analysis and control design applications.

Exact linearization of the specified nonlinear Simulink model produces linear state-space, transfer-function, or zero-pole-gain equations that you can use to:

- Plot the Bode response of the Simulink model.
- Evaluate loop stability margins by computing open-loop response.
- Analyze and compare plant response near different operating points.
- Design linear controller

  Classical control system analysis and design methodologies require linear, time-invariant models. Simulink Control Design automatically linearizes the plant when you tune your compensator. See “Choose a Control Design Approach” on page 8-2.
- Analyze closed-loop stability.
- Measure the size of resonances in frequency response by computing closed-loop linear model for control system.
- Generate controllers with reduced sensitivity to parameter variations and modeling errors.

Linearization in Simulink Control Design

You can use Simulink Control Design to linearize continuous-time, discrete-time, or multirate Simulink models. The resulting linear time-invariant model is in state-space form.

Simulink Control Design uses a *block-by-block* approach to linearize models, instead of using *full-model perturbation*. This block-by-block approach individually linearizes each block in your Simulink model and combines the results to produce the linearization of the specified system.

The block-by-block linearization approach has several advantages to full-model numerical perturbation:

- Most Simulink blocks have preprogrammed linearization that provides Simulink Control Design an exact linearization of each block at the operating point.
- You can configure blocks to use custom linearizations without affecting your model simulation.

  See “When to Specify Individual Block Linearization” on page 2-157.
- Simulink Control Design automatically removes nonminimal states.
- Ability to specify linearization to be uncertain (requires Robust Control Toolbox™)

Model Requirements for Exact Linearization

Exact linearization supports most Simulink blocks.

However, Simulink blocks with strong discontinuities or event-based dynamics linearize (correctly) to zero or large (infinite) gain. Sources of event-based or discontinuous behavior exist in models that have Simulink Control Design requires special handling of models that include:

- Blocks from Discontinuities library
- Stateflow charts
- Triggered subsystems
- Pulse width modulation (PWM) signals

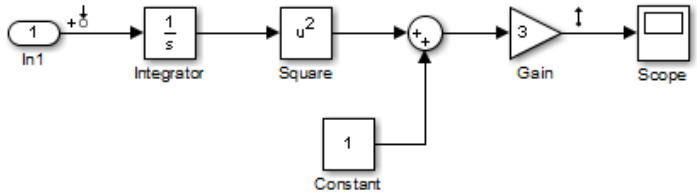
For most applications, the states in your Simulink model should be at steady state. Otherwise, your linear model is only valid over a small time interval.

Operating Point Impact on Linearization

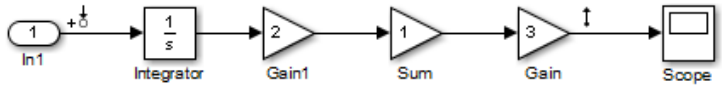
Choosing the right operating point for linearization is critical for obtaining an accurate linear model. The linear model is an approximation of the nonlinear model that is valid only near the operating point at which you linearize the model.

Although you specify which Simulink blocks to linearize, all blocks in the model affect the operating point.

A nonlinear model can have two very different linear approximations when you linearize about different operating points.



The linearization result for this model is shown next, with the initial condition for the integration $x_0 = 0$.



This table summarizes the different linearization results for two different operating points.

Operating Point	Linearization Result
Initial Condition = 5, State `x1` = 5	30/s
Initial Condition = 0, State `x1` = 0	0

You can linearize your Simulink model at three different types of operating points:

- Trimmed operating point — “Linearize at Trimmed Operating Point” on page 2-85
- Simulation snapshot — “Linearize at Simulation Snapshot” on page 2-91
- Triggered simulation event — “Linearize at Triggered Simulation Events” on page 2-95

See Also

More About

- “Exact Linearization Algorithm” on page 2-207
- “Linearize Plant” on page 2-41
- “Linearize Simulink Model at Model Operating Point” on page 2-69
- “Compute Open-Loop Response” on page 2-59
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-77

Choose Linearization Tools

In this section...

Choosing Simulink Control Design Linearization Tools

Simulink Control Design software lets you perform linear analysis of nonlinear models using a user interface, functions, or blocks.

Linearization Tool	When to Use
Linear Analysis Tool	• Interactively explore Simulink model linearization under different operating conditions. • Diagnose linearization problems. • Batch linearize for varying model parameter values. • Automatically generate MATLAB code for batch linearization.
`linearize`	• Linearize a Simulink model for command-line analysis of poles and zeros, plot responses, and control design. • Batch linearize for varying model parameter values and operating points.
`slLinearizer`	Batch linearize for varying model parameter values, operating points, and I/O sets.

Linearization Tool	When to Use
Linear Analysis Plots blocks on page 2-77	• Visualize linear characteristics of your Simulink model during simulation. • View bounds on linear characteristics of your Simulink model on plots. • Optionally, check that the linear characteristics of your Simulink model satisfy specified bounds. **Note** Linear Analysis Plots blocks do not support code generation. You can only use these blocks in `Normal` simulation mode.

Choosing Exact Linearization Versus Frequency Response Estimation

In most cases, to obtaining a linear approximation of a Simulink model, you should use exact linearization instead of frequency response estimation.

Exact linearization:

- Is faster because it does not require simulation of the Simulink model.
- Returns a parametric state-space model.

  Frequency response estimation returns frequency response data. To create a transfer function or a state-space model from the resulting frequency response data, you must fit a model to the data using System Identification Toolbox™ software.

Use frequency response estimation:

- To validate exact linearization accuracy. For more information, see “Validate Linearization In Frequency Domain” on page 2-140.
- When your Simulink model contains discontinuities or non-periodic event-based dynamics.
- To study the impact of amplitude size on frequency response. For more information, see“Describing Function Analysis of Nonlinear Simulink Models”.

Linearization Using Simulink Control Design Versus Simulink

How is Simulink `linmod` different from Simulink Control Design functionality for linearizing nonlinear models?

Although both Simulink Control Design and Simulink `linmod` perform block-by-block linearization, Simulink Control Design functionality is enhanced by a more flexible user interface and Control System Toolbox™ numerical algorithms.

	Simulink Control Design Linearization	Simulink Linearization
Graphical-user interface	Yes See "Linearize Simulink Model at Model Operating Point" on page 2-69.	No
Flexibility in defining which portion of the model to linearize	Yes. Lets you specify linearization I/O points at any level of a Simulink model, either graphically or programmatically without having to modify your model. See "Linearize at Trimmed Operating Point" on page 2-85.	No. Only root-level linearization I/O points, which is equivalent to linearizing the entire model. Requires that you add and configure additional Linearization Point blocks.
Open-loop analysis	Yes. Lets you open feedback loops without deleting feedback signals in the model. See "Compute Open-Loop Response" on page 2-59.	Yes, but requires that you delete feedback signals in your model to open the loop
Control linear model state ordering	Yes See "Order States in Linearized Model" on page 2-130.	No
Control linearization of individual blocks	Yes. Lets you specify custom linearization behavior for both blocks and subsystems. See "When to Specify Individual Block Linearization" on page 2-157.	No

	Simulink Control Design Linearization	Simulink Linearization
Linearization diagnostics	Yes. Identifies problematic blocks and lets you examine the linearization value of each block. See "Linearization Troubleshooting Overview" on page 4-2.	No
Block detection and reduction	Yes. Block reduction detects blocks that do not contribute to the overall linearization yielding a minimal realization.	No
Control of rate conversion algorithm for multirate models	Yes	No

See Also

More About

- "Linearize Nonlinear Models" on page 2-3

Specify Portion of Model to Linearize

In this section...

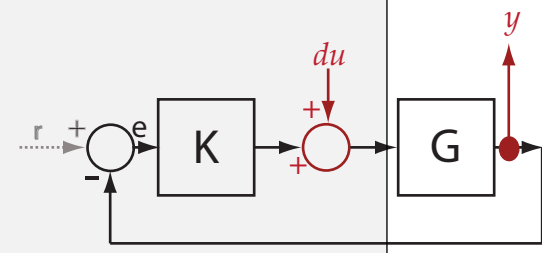
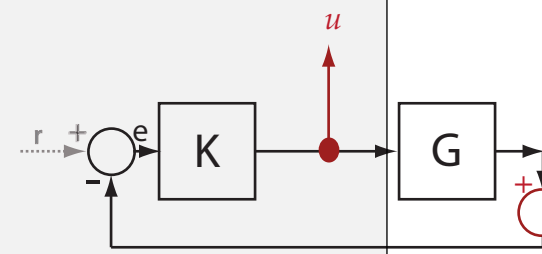
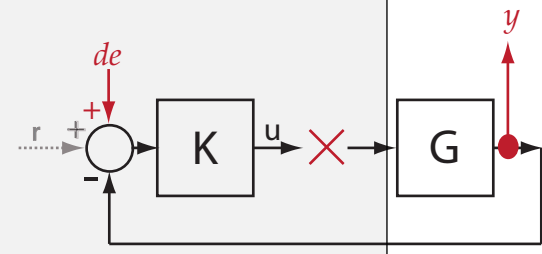
To linearize a subsystem, loop, or block in your model, you use *analysis points*. Each analysis point that you define in the model can serve one or more of the following purposes:

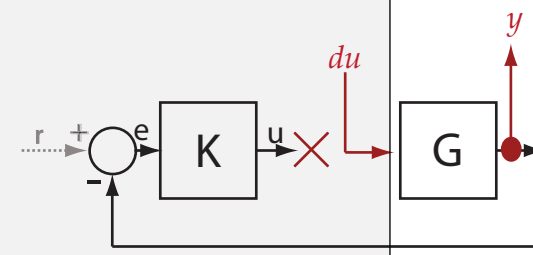
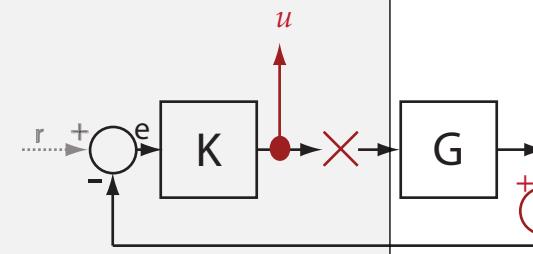
- **Input** — The software injects an additive input signal at an analysis point, for example, to model a disturbance at the plant input.
- **Output** — The software measures the signal value at a point, for example, to study the impact of a disturbance on the plant output.
- **Loop Opening** — The software interprets a break in the signal flow at a point, for example, to study the open-loop response at the plant input.

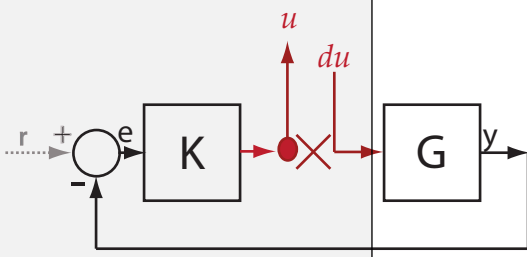
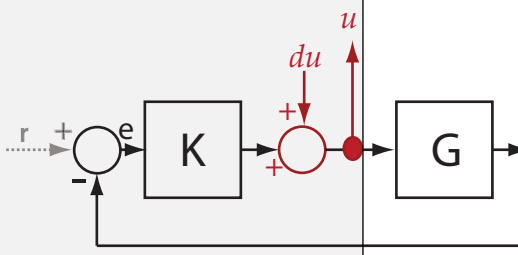
To compute a linear model for a portion of your system, specify a linearization input point and output point on the input and output signal to the portion of the model you want to linearize. To compute an open-loop response, specify loop openings to break the signal flow. You can also compute MIMO linear models by defining multiple input and output points.

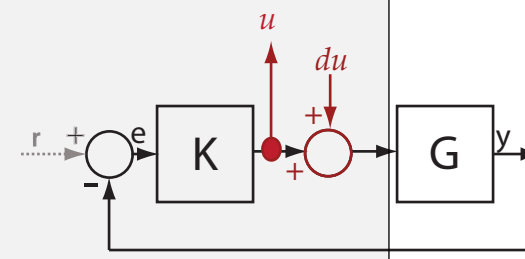
Analysis Points

You can specify the following types of linear analysis points using Simulink Control Design software. These analysis points are pure annotations and do not impact model simulation.

Analysis Point	Description	Example System
Input perturbation	Specifies an additive input to a signal. To define a transfer function for a linearized system, you can use an input perturbation with an output measurement or open-loop output. For example, to compute the response $G/(1+GK)$ in the example system, specify an input perturbation du and an output measurement y as shown.	
Output measurement	Takes a measurement at a signal. To define a transfer function for a linearized system, you can use an output measurement with an input perturbation or an open-loop input. For example, to compute the response $-K/(1+KG)$ in the example system, specify an output measurement point u and an input perturbation dy as shown.	
Loop break	Specifies a loop opening. Use a loop break to compute open-loop transfer function around a loop. Typically, you use loop breaks when you have nested loops or want to ignore the effect of some loops. In the example system, the loop break stops the signal flow at u. As a result, the transfer function from the input perturbation de to the output measurement y is 0.	

Analysis Point	Description	Example System
Open-loop input	Specifies a loop break followed by an input perturbation. To linearize a plant or controller, you can use an open-loop input with an output measurement or an open-loop output. For example, to linearize the plant in the example system, add an open-loop input before *G* and an output measurement *y* after *G*, as shown. The open-loop input breaks the signal flow at *u*, and adds an input perturbation *du*.	
Open-loop output	Specifies an output measurement followed by a loop break. To linearize a plant or controller, you can use an open-loop output with an input perturbation or an open-loop input. For example, to compute the response -*K* in the example system, add an open-loop output after *K* and an input perturbation *dy* after *G*, as shown. The open-loop output breaks the signal flow and adds an output measurement *u*.	

Analysis Point	Description	Example System
Loop transfer function	Specifies an output measurement before a loop break followed by an input perturbation. To compute the open-loop transfer function around a loop, use a loop transfer analysis point. For example, to compute -*KG* in the example system, specify the loop transfer analysis point as shown. The software adds an output measurement *u* breaks the signal flow, and adds an input perturbation *du*.	
Sensitivity function	Specifies an input perturbation followed by an output measurement. The sensitivity function measures how sensitive a signal is to an added disturbance. Sensitivity is a closed-loop measure. Feedback reduces the sensitivity in the frequency band where the open-loop gain is greater than `1`. For example, to compute the sensitivity at the plant input of the example system, add a sensitivity function analysis point as shown. The software adds an input perturbation *du* followed by an output measurement *u*. The closed-loop transfer function from *du* to *u* is 1/(1+*GK*).	

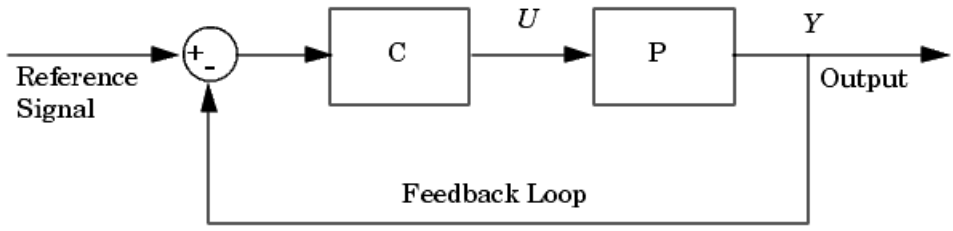
Analysis Point	Description	Example System
 Complementary sensitivity function	Specifies an output measurement followed by an input perturbation. The complementary sensitivity function at a point is the transfer function from an additive disturbance at the point to a measurement at the same point. In contrast to the sensitivity function, the disturbance is added after the measurement. Use this analysis point to compute closed-loop transfer function around the loop. For example, to compute the closed-loop transfer function for the example system, add a complementary sensitivity function analysis point as shown. The software adds an output measurement *u* followed by and input perturbation *du*. The closed-loop transfer function from *du* to *u* is *GK*/(1+*GK*).	

Opening Feedback Loops

If your model contains one or more feedback loops, you can choose to linearize an open-loop or a closed-loop system.

To remove the effects of a feedback loop, using analysis points lets you insert a loop opening without manually breaking the signal line. Manually removing the feedback signal from a nonlinear model changes the model operating point and produces a different linearized model. For more information, see “How the Software Treats Loop Openings” on page 2-39.

Proper placement of the loop opening is important for obtaining the linear model that you want. To understand the difference between open-loop and closed-loop analysis, consider the following single-loop control system.

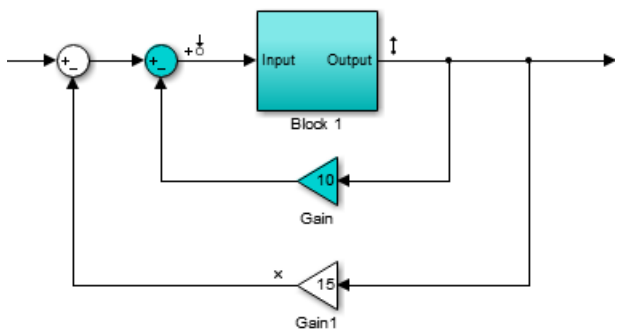


Suppose that you want to linearize the plant P about an equilibrium operating point of the model.

To linearize only the plant, you open the loop at the output of block P. If you do not open the loop, the linearized model between U and Y includes the effect of the feedback loop.

Loop open at Y?	Transfer Function from U to Y
Yes	$P(s)$
No	$\frac{P(s)}{1+P(s)C(s)}$

The loop opening does not have to be in the same location as the linearization input or output point. For example, the following system has a loop opening after the gain on the outer feedback loop, which removes the effect of this loop from the linearization. As a result, only the blue blocks are on the linearization path.



In this example, if you place a loop opening at the same location as the linearization output point, the effect of the inner loop is also removed from the linearization result.

Ways to Specify Portion of Model to Linearize

There are several ways to define the portion of the model you want to linearize using linear analysis points. Each method has its own advantages and depends on which linearization tool you use. For more information on choosing linearization tools, see "Choose Linearization Tools" on page 2-9.

Specify portion of model...	Use this method if...	For more Information, see...
In Simulink model	You want to save the analysis points directly in the model or graphically display the analysis points within the model.	"Specify Portion of Model to Linearize in Simulink Model" on page 2-21
Using Linear Analysis Tool	You want to linearize your model interactively using the Linear Analysis Tool without changing the Simulink model. Using this method you can specify multiple open-loop or closed-loop transfer functions for your model.	"Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29
At command line using `linio` command	You want to linearize your model using the `linearize` command. Using `linio` does not change the Simulink model.	`linio`, `getlinio`, and `setlinio`
Using `slLinearizer` interface	You want to obtain multiple open-loop or closed-loop transfer functions from the linearized system without recompiling the model. Using this method does not change the Simulink model.	"Mark Signals of Interest for Batch Linearization" on page 3-13

Specify portion of model...	Use this method if...	For more Information, see...
Using `slTuner` interface	You want to obtain multiple open-loop or closed-loop transfer functions from a tuned control system without recompiling the model. Using this method does not change the Simulink model.	"Mark Signals of Interest for Control System Analysis and Design" on page 2-48
As a specific block or subsystem	You want to linearize a specific block or subsystem without defining analysis points for all the block inputs and outputs. Using this method does not change the Simulink model.	"Linearize Plant" on page 2-41

See Also

`linearize` | `linio` | `slLinearizer` | `slTuner`

More About

- "Choose Linearization Tools" on page 2-9
- "Linearize Simulink Model at Model Operating Point" on page 2-69
- "Compute Open-Loop Response" on page 2-59

Specify Portion of Model to Linearize in Simulink Model

In this section...

To specify the portion of the model to linearize, you can define and save linear analysis points directly in your Simulink model. Analysis points represent linearization inputs, outputs, and loop openings for your model.

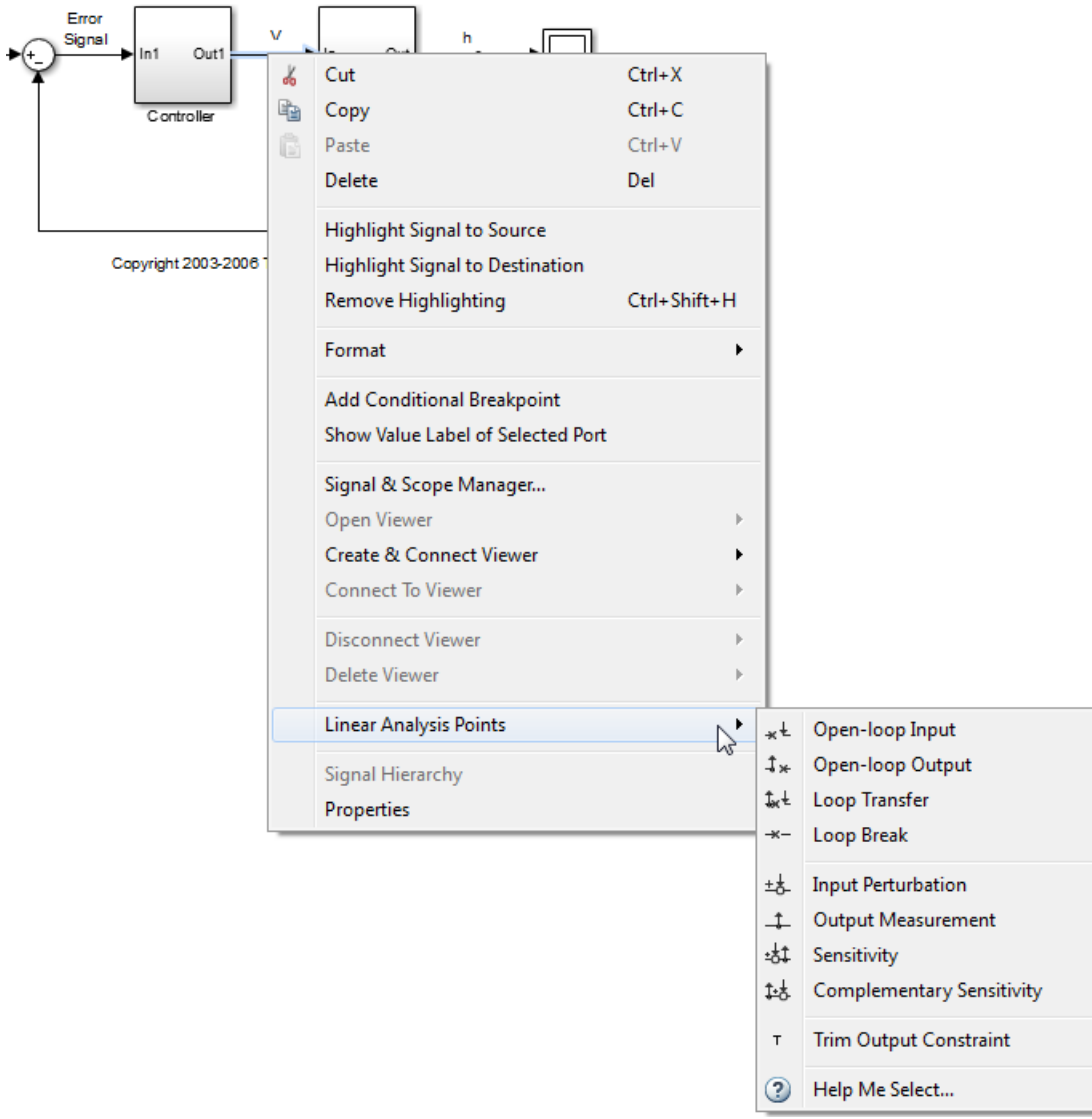
Alternatively, to specify analysis points without changing your model, you can define analysis points:

- In the Linear Analysis Tool. For more information, see “Specify Portion of Model to Linearize in Linear Analysis Tool” on page 2-29.
- At the command line. For more information, see `linio`, `getlinio`, and `setlinio`.

Specify Analysis Points

To specify analysis points directly in your Simulink model:

1 Right-click the signal you want to define as an analysis point, and hover the cursor over **Linear Analysis Points**.



2 Under **Linear Analysis Points**, select the type of analysis point you want to define.

- **Input Perturbation** — Specifies an additive input to a signal.

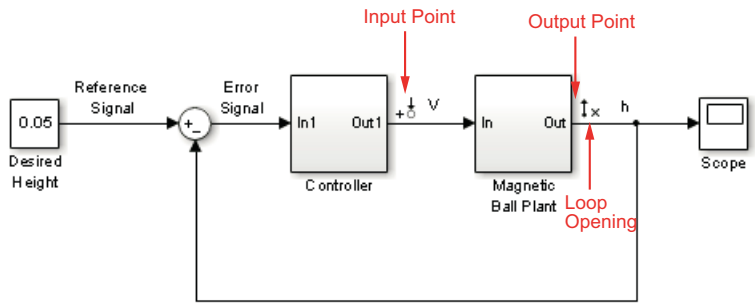
- **Output Measurement** — Takes a measurement at a signal.
- **Loop Break** — Specifies a loop opening.
- **Open-Loop Input** — Specifies a loop break followed by an input perturbation.
- **Open-Loop Output** — Specifies an output measurement followed by a loop break.
- **Loop Transfer** — Specifies an output measurement before a loop break followed by an input perturbation.
- **Sensitivity** — Specifies an input perturbation followed by an output measurement.
- **Complementary Sensitivity** — Specifies an output measurement followed by an input perturbation.

For more information on the different types of analysis points, see "Specify Portion of Model to Linearize" on page 2-13.

When you specify analysis points, the software adds annotations to your model indicating the linear analysis point type.

3 Repeat steps 1 and 2 for all signals you want to define as analysis points.

For each linear analysis point that you specify, the software adds an annotation to your model indicating the analysis point type.

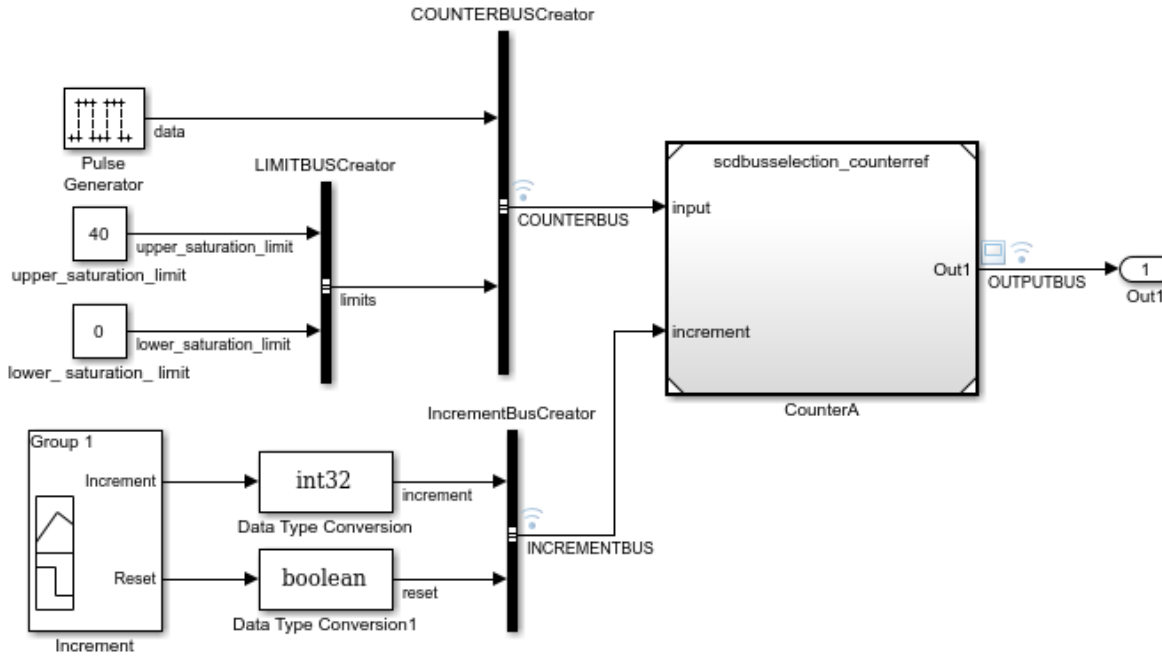


Select Bus Elements as Analysis Points

This example shows how to select individual elements in a bus signal as analysis points.

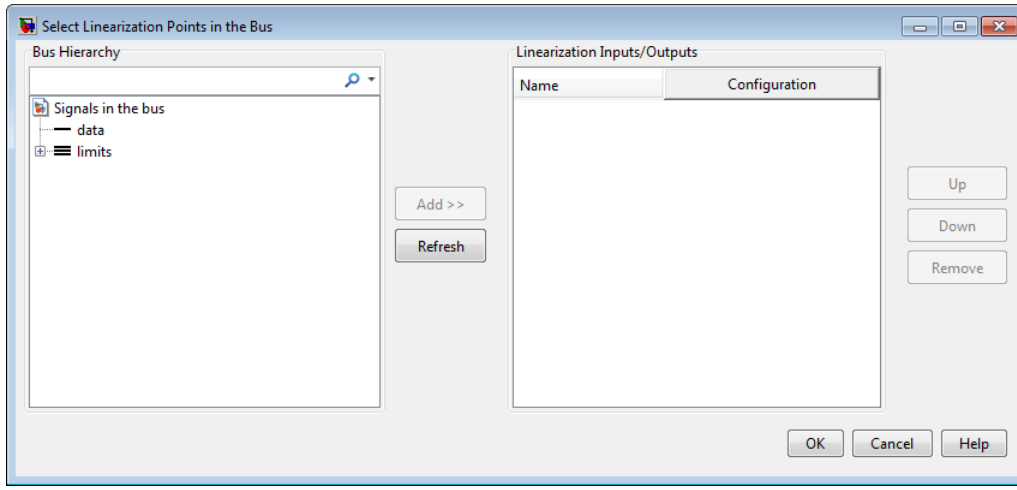
1 Open Simulink model.

```
sys = 'scdbusselection';
open_system(sys)
```

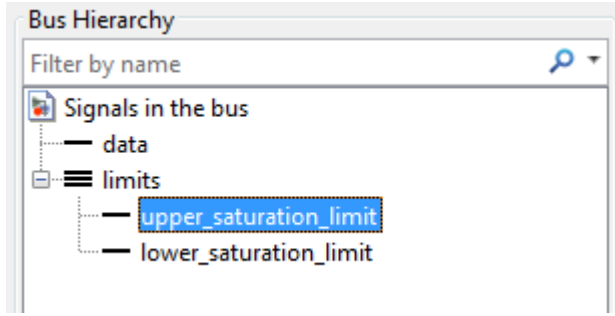


2 Specify a bus signal as a linear analysis point.

In the Simulink model window, right-click a bus signal, such as the `OUTPUTBUS` signal, and select **Linear Analysis Points > Select Bus Element**.



In the Select Linearization Points in the Bus dialog box, in the **Bus Hierarchy** section, expand the `limits` bus, and select `upper_saturation_limit`. `limits` is a nested bus within the `OUTPUTBUS` signal.



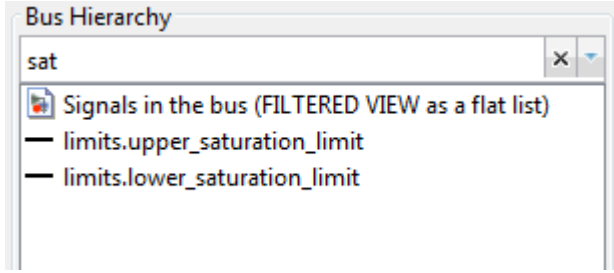
Tip To filter bus elements by name within a large bus, you can enter search text in the **Filter by name** box. The name match is case-sensitive. Also, you can enter a MATLAB regular expression (MATLAB).

To modify the filtering options, click  next to the **Filter by name** box.

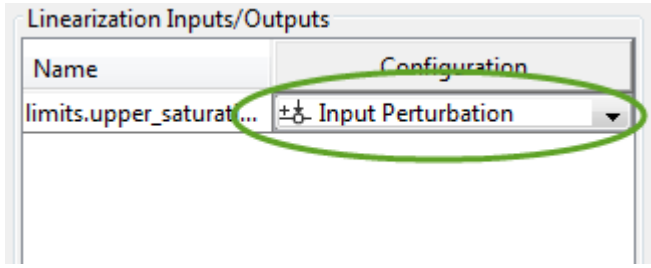
Filtering Options

You can specify the following foptions when filtering the list of bus signals.

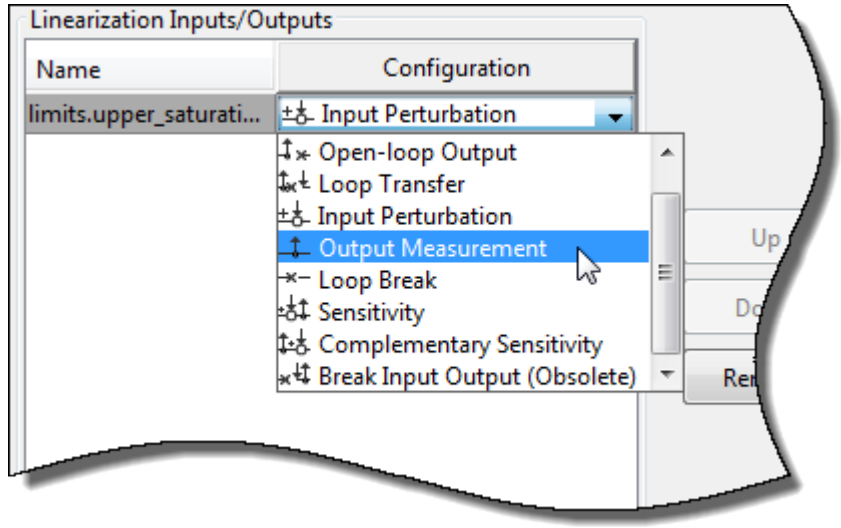
- **Enable regular expression** — Use MATLAB regular expressions for filtering signal names. For example, entering `t$` displays all signals whose names end with a lowercase `t` (and their immediate parents).
- **Show filtered results as a flat list** — Display the filtered signals in a flat list. By default, filtered signals are displayed using a tree format. The flat list format uses dot notation to reflect the hierarchy of bus signals.



To add the selected signal to the **Linearization Inputs/Outputs** section, click **Add**. By default, the signal is configured as an `Input Perturbation` analysis point.



You can change the analysis point type using the **Configuration** drop-down list. For example, to specify a linearization output point, select `Output Measurement`.



3 To add additional analysis points from within the same bus signal, repeat step 2.

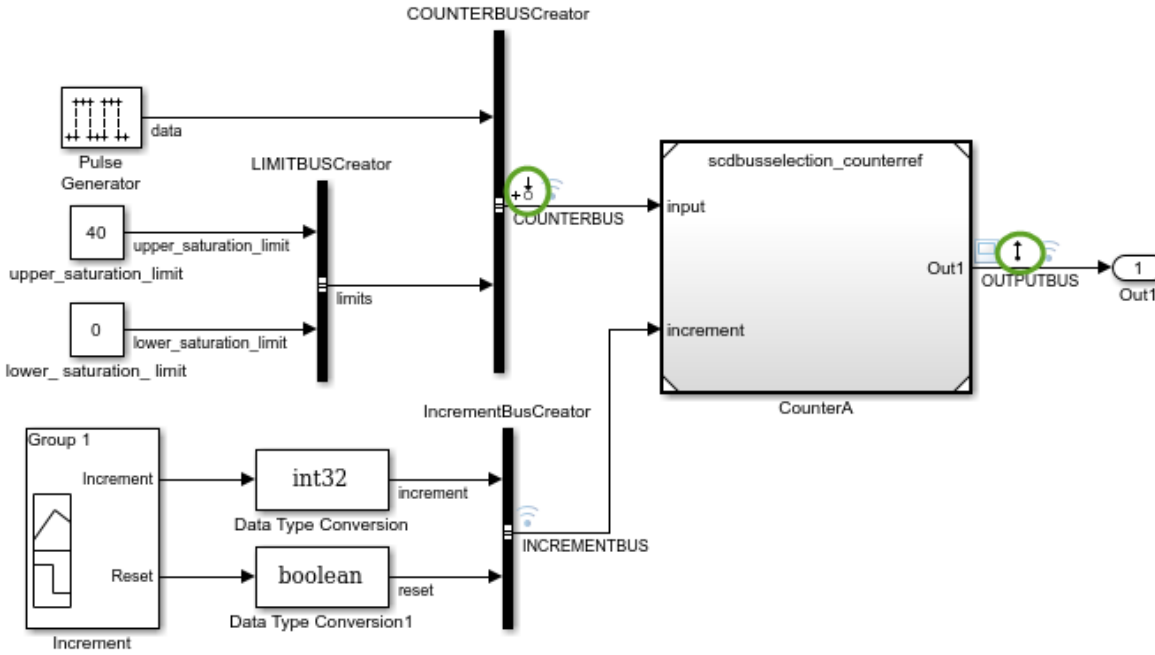
To remove an analysis point, select the signal in the **Linearization Inputs/ Outputs** section, and click **Remove**.

Once you have defined all of the required analysis points for that bus, click **OK**.

4 To specify additional analysis points

5 To view linear analysis point indicators in the Simulink model, in the model window, select **Display** > **Signals & Ports** > **Linearization Indicators**.

The software adds graphical annotations to the bus signals indicating the type of analysis points specified. For example, if you specify a linearization input in the `COUNTERBUS` signal and a linearization output in the `OUTPUTBUS` signal, the software adds the corresponding annotations to the signals.



You can specify different analysis point types for multiple elements in the same bus. In this case, the software adds the annotation to the signal.

See Also

More About

- "Specify Portion of Model to Linearize" on page 2-13
- "Linearize Simulink Model at Model Operating Point" on page 2-69
- "Compute Open-Loop Response" on page 2-59

Specify Portion of Model to Linearize in Linear Analysis Tool

To specify the portion of the model to linearize, you can define linear analysis points using the Linear Analysis Tool. Analysis points represent linearization inputs, outputs, and loop openings for your model. Using this method you can specify multiple sets of analysis points without changing your model.

Alternatively, you can define analysis points:

- Programmatically at the command line. For more information, see `linio`, `getlinio`, and `setlinio`.
- Directly in your Simulink model. Use this method to save your analysis points in the model. For more information, see "Specify Portion of Model to Linearize in Simulink Model" on page 2-21.

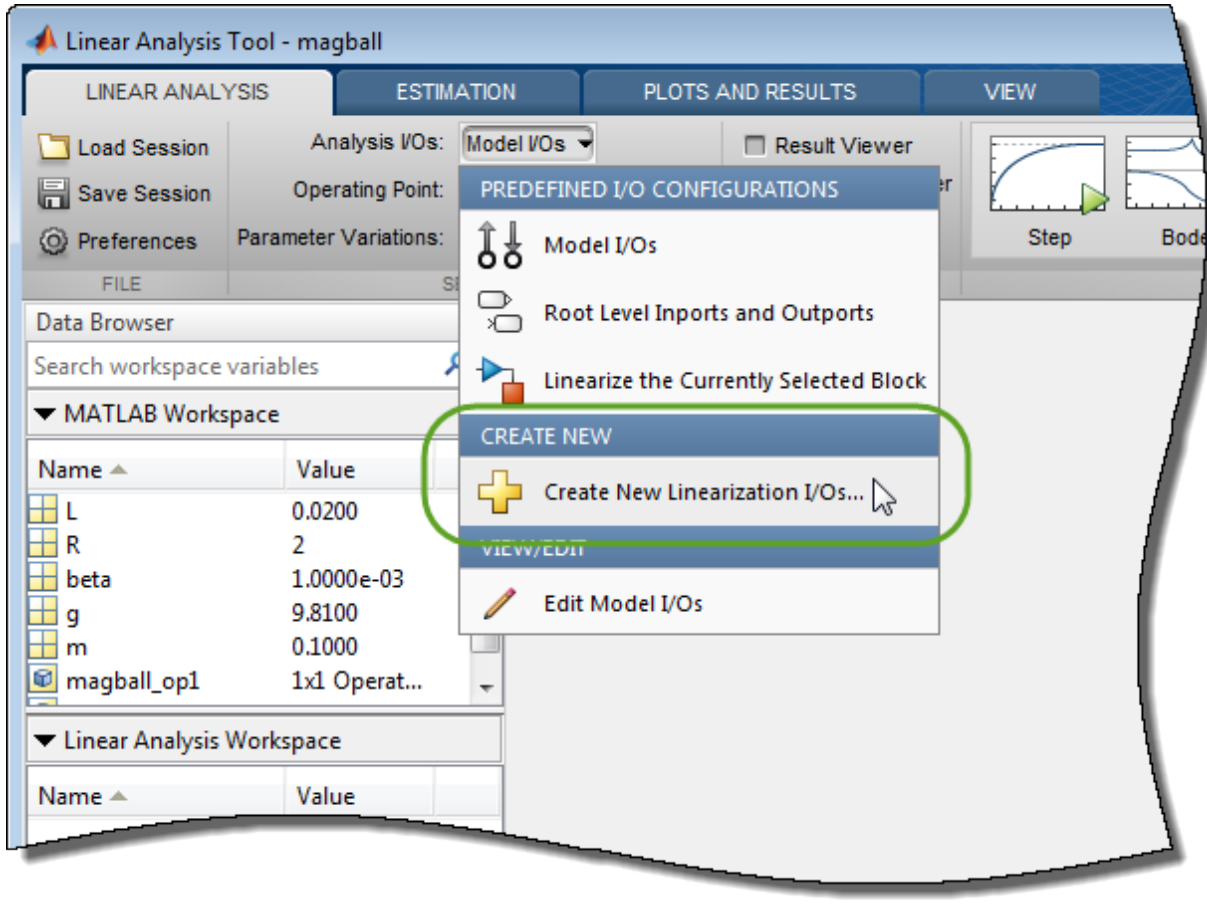
In this section...

Specify Analysis Points

In the Linear Analysis Tool, you specify analysis points using linearization I/O sets. You can specify one or more linearization I/O sets, without introducing changes to the model.

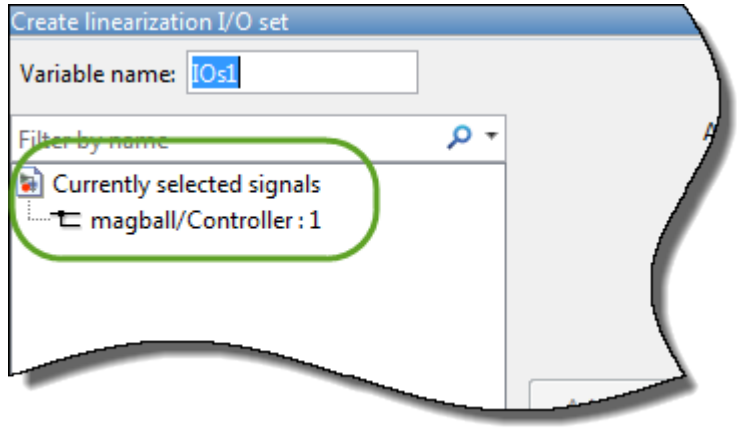
To create a linearization I/O set:

1 On the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Create New Linearization I/Os`.



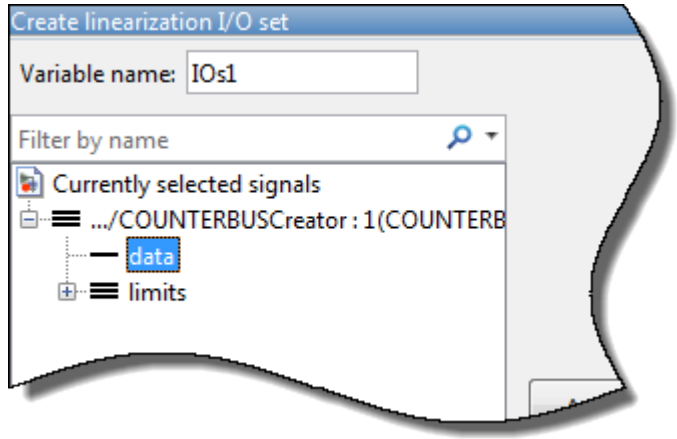
1 In your Simulink model, select one or more signals that you want to define as analysis points.

   The selected signals appear in the Create linearization I/O set dialog box under **Currently selected signals**.

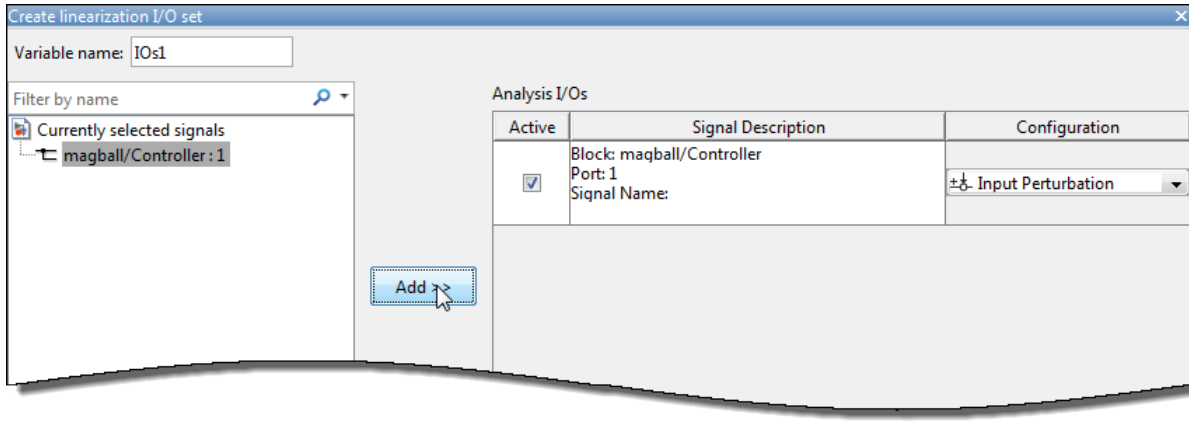


2 Under **Currently selected signals**, click the signal you want to add. To select multiple signals, hold **Ctrl** and click each signal you want to add.

To add a signal from within a bus signal, expand the bus and select the signal. For example, select the `data` signal within the `COUNTERBUS` signal.



3 To add the signal to list of **Analysis I/Os**, click **Add**.



4 In the **Configuration** drop-down list for the signal, select the type of analysis point you want to define:

- **Input Perturbation** — Specifies an additive input to a signal.
- **Output Measurement** — Takes a measurement at a signal.
- **Loop Break** — Specifies a loop opening.
- **Open-Loop Input** — Specifies a loop break followed by an input perturbation.
- **Open-Loop Output** — Specifies an output measurement followed by a loop break.
- **Loop Transfer** — Specifies an output measurement before a loop break followed by an input perturbation.
- **Sensitivity** — Specifies an input perturbation followed by an output measurement.
- **Complementary Sensitivity** — Specifies an output measurement followed by an input perturbation.

For more information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-13.

5 Repeat steps 1–4 for any other signals you want to define as analysis points.

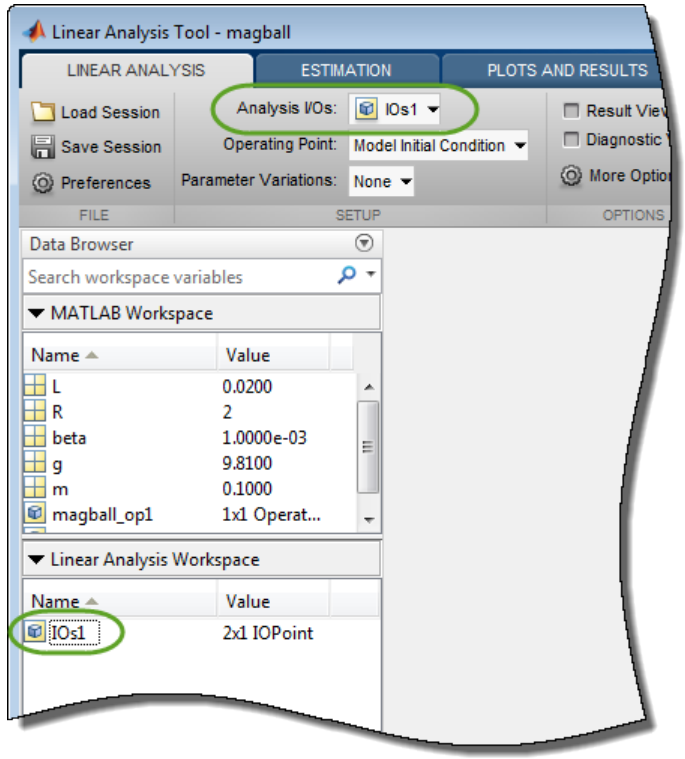
Tip To highlight the source block of an analysis point in the Simulink model, in the **Analysis I/Os** list, select the analysis point, and click **Highlight**.

6 In the **Variable name** box, enter a name for the I/O set.

7 Click **OK**.

The software adds the linearization I/O set to the **Linear Analysis Workspace**.

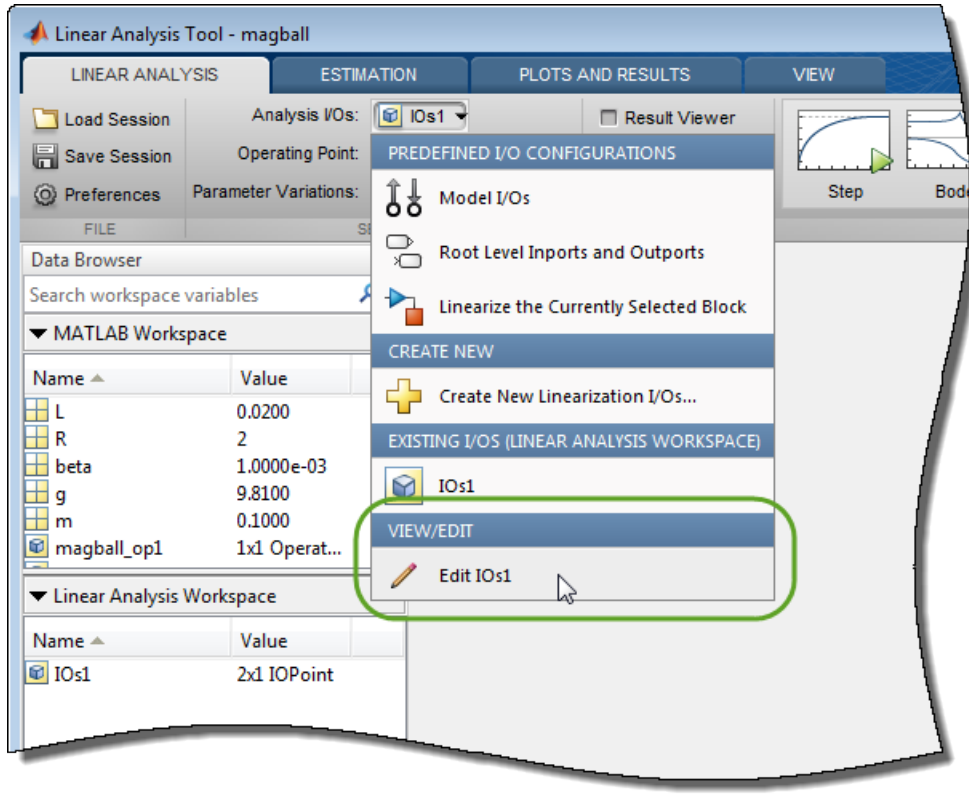
The software also adds the linearization I/O set to the **Analysis I/Os** drop-down list and automatically selects it.



Edit Analysis Points

You can interactively edit a linearization I/O set stored in the Linear Analysis Tool using the Edit dialog box. To open the Edit dialog box, in the **Linear Analysis Workspace**, double-click the I/O set you want to edit.

Alternatively, you can open the Edit dialog box for the current selected linearization I/O set in the **Analysis I/Os** drop-down list. To do so, in the drop-down list, under **View/Edit**, click `Edit`.



In the Edit dialog box, you can add or remove analysis points, change the type for existing analysis points, or enable or disable analysis points. Once you have finished editing the I/O set, save your changes by closing the dialog box.

Tip To highlight the location in the Simulink model of any signal in the current list of analysis I/O points, select the I/O point in the list, and click **Highlight**.

Add Analysis Point to I/O Set

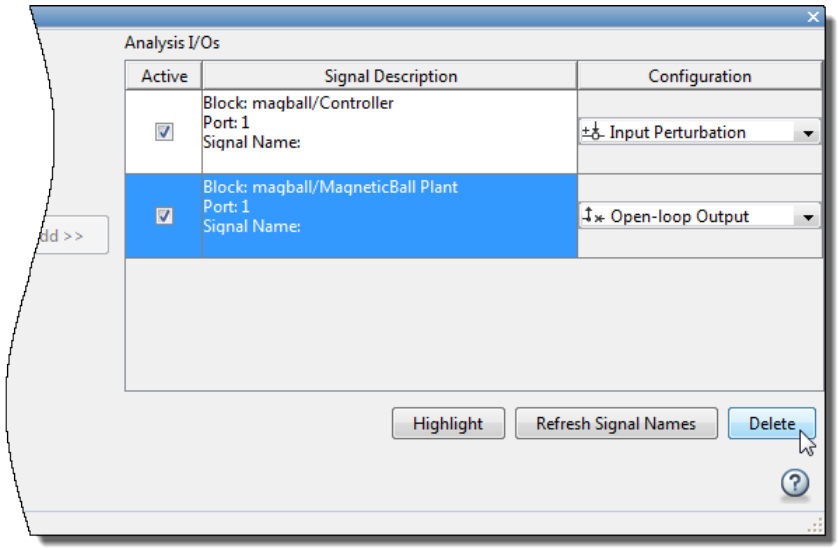
To add an analysis point to the linearization I/O set:

1. In your Simulink model, select one or more signals that you want to add to the linearization I/O set.

   The selected signals appear in the Edit dialog box under **Currently selected signals**.
2. Under **Currently selected signals**, click the signal you want to add. To select multiple signals, hold **Ctrl**, and click each signal you want to add.
3. To add the signal to list of **Analysis I/Os**, click **Add**.
4. In the **Configuration** drop-down list for the signal, select the type of analysis point you want to define. For example, if you want the signal to be an open-loop linearization output point, select `Open-loop Output`.

Remove Analysis Point from I/O Set

To remove an analysis point from the linearization I/O set, in the **Analysis I/Os** section, click the signal you want to remove, and click **Delete**.



Change Analysis Point Type

To change the linear analysis point type for a signal, in the **Analysis I/Os** section, in the **Configuration** drop-down list for the signal, select the analysis point type. For example, if you want the signal to be a linearization output point, select `Output Measurement`.

Enable or Disable Analysis Points

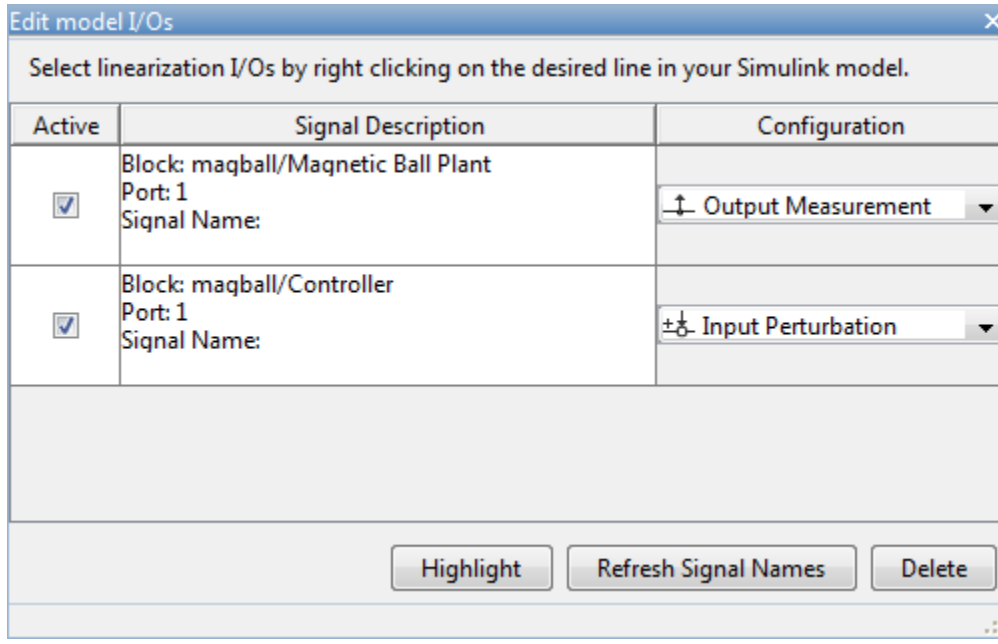
To modify an existing linearization I/O set without removing analysis points, you can disable one or more analysis points. To do so, in the **Analysis I/Os** section, under **Active**, clear the corresponding check box.

When you linearize your model using the linearization I/O set, the software ignores any disabled analysis points.

To enable a disabled analysis point, select the corresponding check box.

Edit Simulink Model Analysis Points

You can modify analysis points stored in your Simulink model using the Linear Analysis Tool. To do so, on the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Model I/Os`, and then, in same drop-down list, select `Edit Model I/Os`.



In the Edit model I/Os dialog box, you can:

- Change the type for an analysis point using the corresponding **Configuration** drop-down list.
- Delete an analysis point from the model. To do so, click the signal you want to remove, and click **Delete**.
- Enable or disable an analysis point using the corresponding **Active** check box. When you disable an analysis point, in the Simulink model, the software removes the annotation from the corresponding signal.

Note If you close the Linear Analysis Tool, any analysis points that you disabled in this manner are deleted from the Simulink model. To keep the analysis points in the model, reenable them before closing the Linear Analysis Tool.

For information on adding analysis points to the model, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-21.

See Also

Linear Analysis Tool

More About

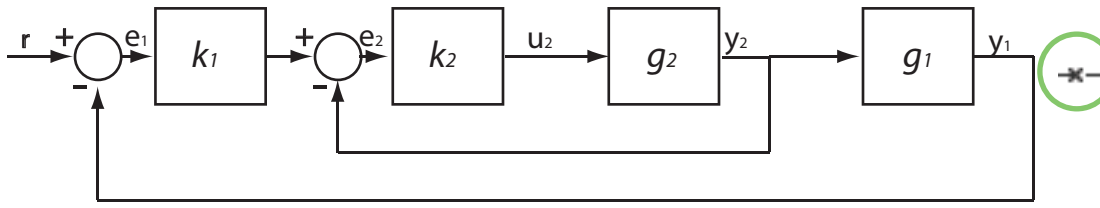
- “Specify Portion of Model to Linearize” on page 2-13
- “Linearize Simulink Model at Model Operating Point” on page 2-69
- “Compute Open-Loop Response” on page 2-59

How the Software Treats Loop Openings

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations. For more information, see "Exact Linearization Algorithm" on page 2-207.

To obtain an open-loop transfer function from a model, you specify a loop opening. Loop openings affect only how the software recombines the individual linearized blocks. In other words, the software ignores loop openings when determining the input signal levels for each block, which affects how nonlinear blocks are linearized.

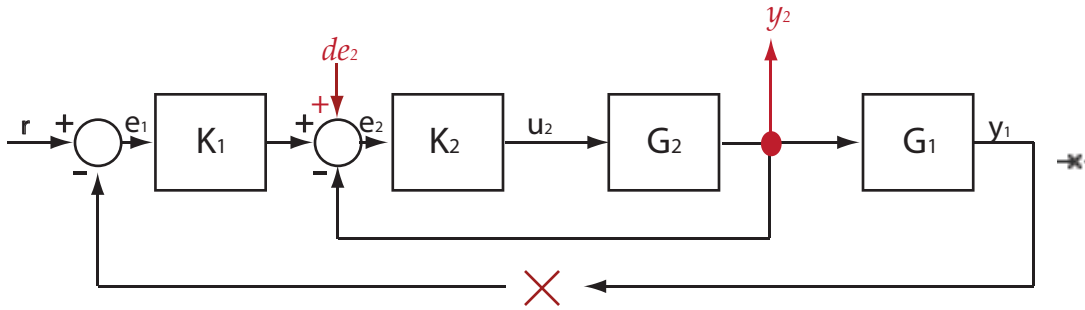
For example, in the following model, to compute the response from e_2 to y_2 without the effects of the outer loop, you open the outer loop by placing a loop opening analysis point at y_1.



Here, k_1, k_2, g_1, and g_2 are nonlinear blocks.

The software linearizes each individual block at the specified operating point, creating the linearized blocks K_1, K_2, G_1, and G_2. At this stage, the software does not break the signal flow at y_1. Therefore, the block linearizations include the effects of the inner-loop and outer-loop feedback signals.

To compute the transfer function from e_2 to y_2, the software enforces the loop opening at y_1, injects an input signal at e_2, and measures the output at y_2.



Here, K_1, K_2, G_1, and G_2 are the linearized blocks.

The resulting linearized transfer function is $(I+G_2K_2)^{-1}G_2K_2$.

See Also

`addOpening` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize`

More About

- “Specify Portion of Model to Linearize” on page 2-13
- “Mark Signals of Interest for Batch Linearization” on page 3-13
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-48
- “Compute Open-Loop Response” on page 2-59

Linearize Plant

You can linearize a block or subsystem in your Simulink model without defining separate analysis points for the block inputs and outputs. The software isolates the selected block from the rest of the model and computes a linear model of the block from the block inputs to the block outputs.

Linearizing a block in this way is equivalent to specifying open-loop input and open-loop output analysis points at the block inputs and outputs, respectively. For more information on specifying analysis points in your model, see “Specify Portion of Model to Linearize” on page 2-13.

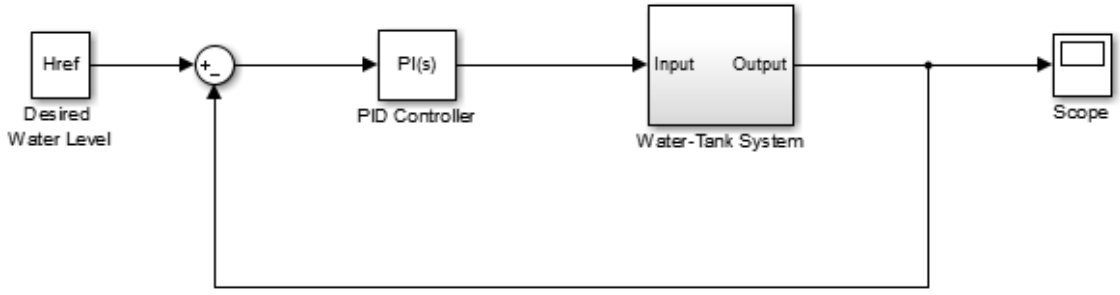
In this section...

Linearize Plant Using Linear Analysis Tool

This example shows how to linearize a plant subsystem in a Simulink model using the **Linear Analysis Tool**.

Open Simulink model.

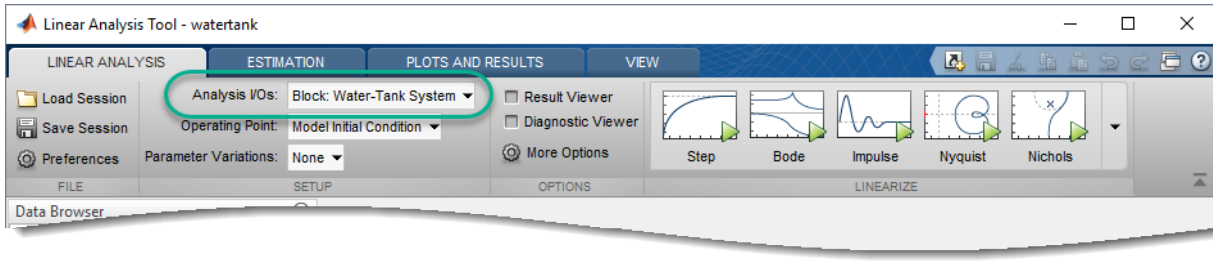
```
mdl = 'watertank';
open_system(mdl)
```



For this system, the Water-Tank System block contains all the nonlinear dynamics. To linearize this system, open the Linear Analysis Tool, and select the block as a linearization I/O set.

To open the Linear Analysis Tool, in the Simulink model window, right-click the Water-Tank System block, and select **Linear Analysis > Linearize Block**.

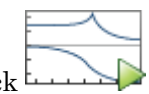
In the Linear Analysis Tool, on the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, the software sets the I/O set for linearization to `Block: Water-Tank System`.



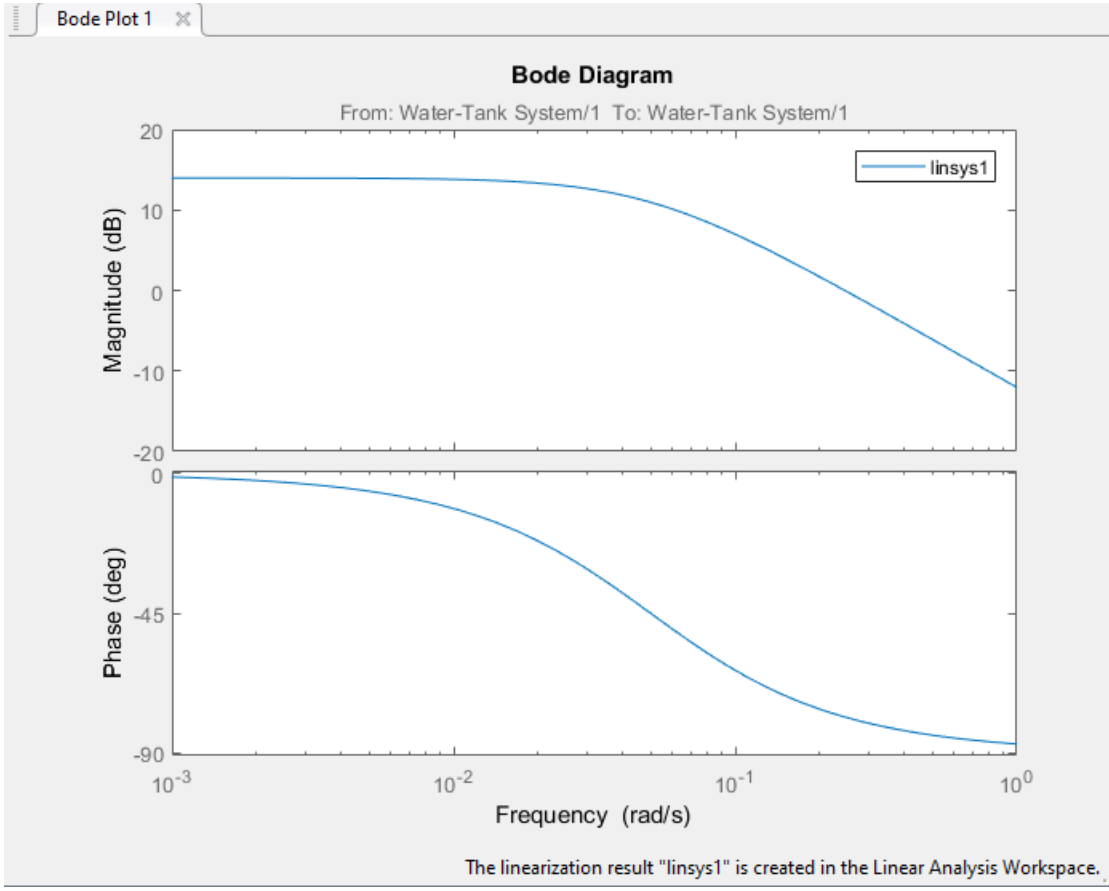
Alternatively, if the Linear Analysis Tool is already open for your system, in the Simulink model window, click the Water-Tank System block. Then, in the Linear Analysis Tool, in the **Analysis I/Os** drop-down list, select `Linearize the Currently Selected Block`.

Tip When the specified linearization I/O set is a block, you can highlight the block in the model by selecting the view option from the **Analysis I/Os** drop-down list. For example, to highlight the Water-Tank System block, select `View Water-Tank System`.

For this example, use the model operating point for linearization. The model operating point consists of the initial state values and input signals stored in the model. In the Linear Analysis Tool, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, leave `Model Initial Condition` selected. For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-85 and “Linearize at Simulation Snapshot” on page 2-91.

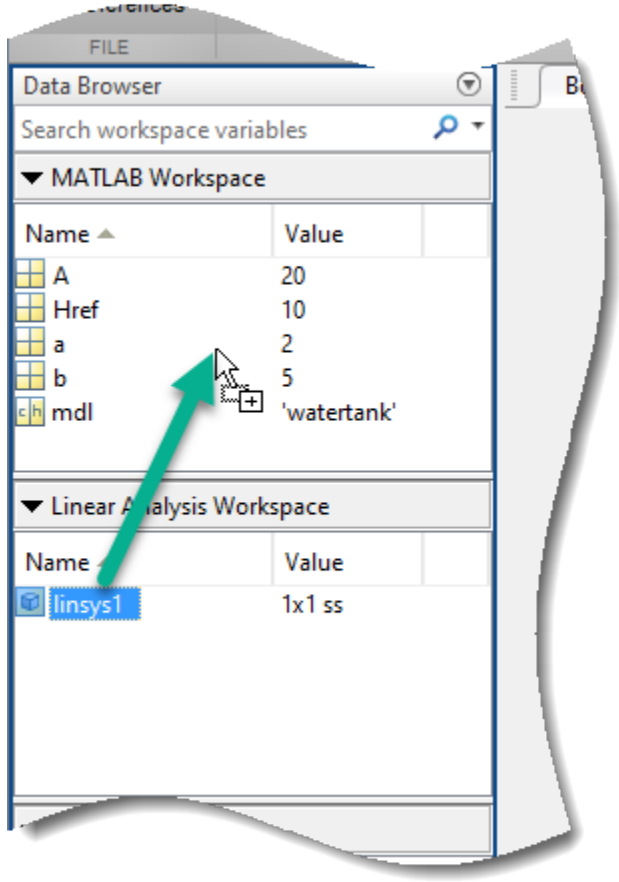
To linearize the specified block and generate a Bode plot for the resulting linear model, click  **Bode**.

The software adds the linearized model, `linsys1`, to the **Linear Analysis Workspace** and generates a Bode plot for the model.



For more information on analyzing linear models, see “Analyze Results Using Linear Analysis Tool Response Plots” on page 2-146.

You can also export the linearized model to the MATLAB workspace. To do so, in the **Data Browser**, drag `linsys1` from the **Linear Analysis Workspace** to the **MATLAB Workspace**.

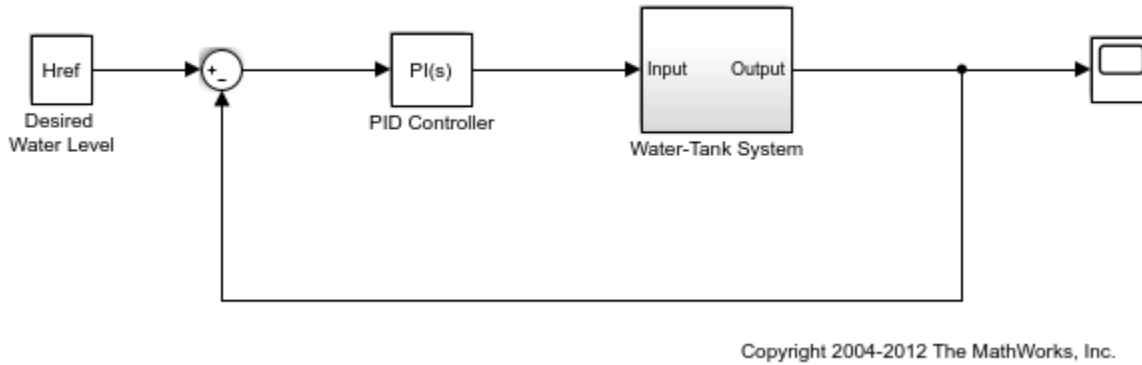


Linearize Plant at Command Line

This example shows how to linearize a plant subsystem in a Simulink® model using the `linearize` command.

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



For this system, the Water-Tank System block contains all the nonlinear dynamics. To linearize this subsystem, first specify its block path.

```
blockpath = 'watertank/Water-Tank System';
```

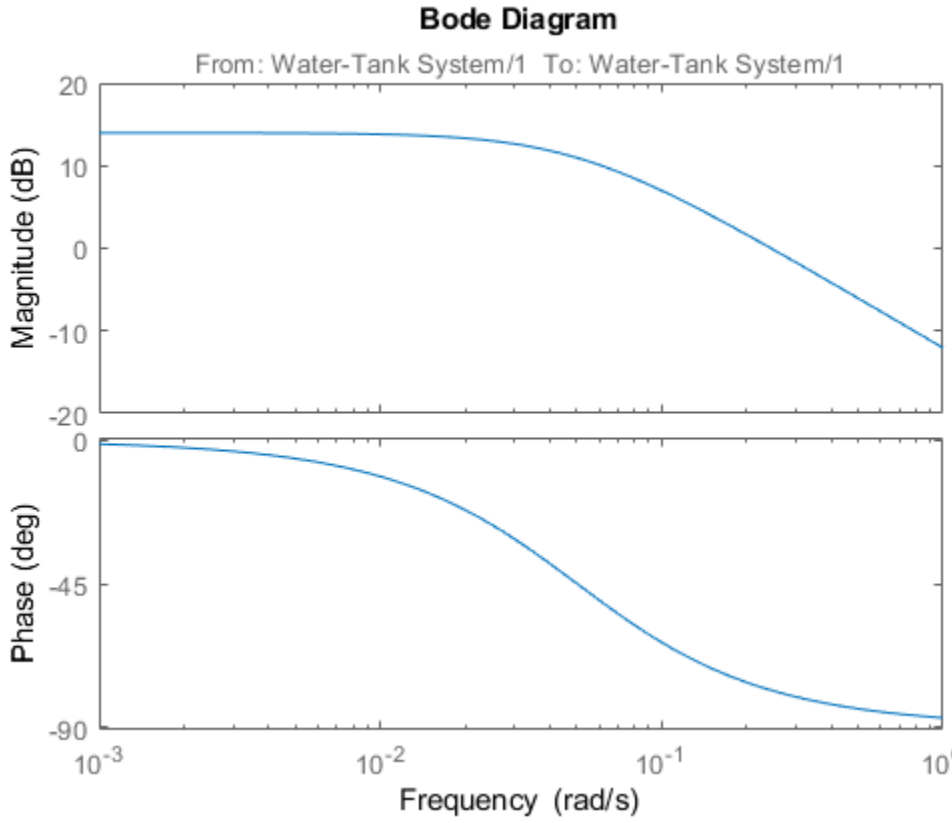
Then, linearize the plant subsystem at the model operating point.

```
linsys1 = linearize(mdl,blockpath);
```

The model operating point consists of the initial state values and input signals stored in the model. For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-85 and “Linearize at Simulation Snapshot” on page 2-91.

You can then analyze the response of the linearized model. For example, plot its Bode response.

```
bode(linsys1)
```



For more information on analyzing linear models, see “Linear Analysis” (Control System Toolbox).

See Also

Linear Analysis Tool | `linearize`

More About

- “Linearize Simulink Model at Model Operating Point” on page 2-69
- “Compute Open-Loop Response” on page 2-59

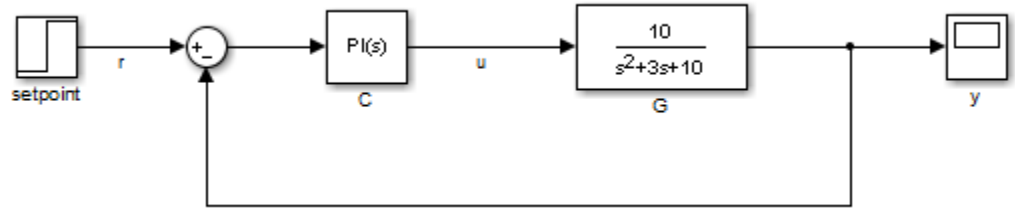
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-77

Mark Signals of Interest for Control System Analysis and Design

In this section...

Analysis Points

Whether you model your control system in MATLAB or Simulink, use analysis points to mark points of interest in the model. Analysis points allow you to access internal signals, perform open-loop analysis, or specify requirements for controller tuning. In the block diagram representation, an analysis point can be thought of as an access port to a signal flowing from one block to another. In Simulink, analysis points are attached to the outports of Simulink blocks. For example, in the following model, the reference signal, `r`, and the control signal, `u`, are analysis points that originate from the outputs of the setpoint and C blocks respectively.

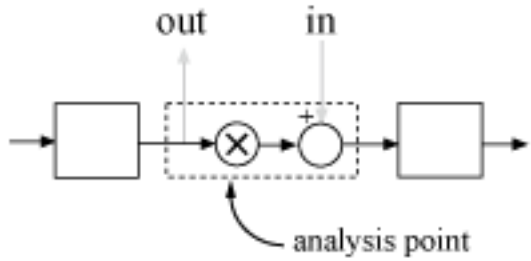


Each analysis point can serve one or more of the following purposes:

- **Input** — The software injects an additive input signal at an analysis point, for example, to model a disturbance at the plant input.
- **Output** — The software measures the signal value at a point, for example, to study the impact of a disturbance on the plant output.
- **Loop Opening** — The software inserts a break in the signal flow at a point, for example, to study the open-loop response at the plant input.

You can apply these purposes concurrently. For example, to compute the open-loop response from `u` to `y`, you can treat `u` as both a loop opening and an input. When you use

an analysis point for more than one purpose, the software applies the purposes in this sequence: output measurement, then loop opening, then input.



Using analysis points, you can extract open-loop and closed-loop responses from a control system model. For example, suppose `T` represents the closed-loop system in the model above, and `u` and `y` are marked as analysis points. `T` can be either a generalized state-space model or an `slLinearizer` or `slTuner` interface to a Simulink model. You can plot the closed-loop response to a step disturbance at the plant input with the following commands:

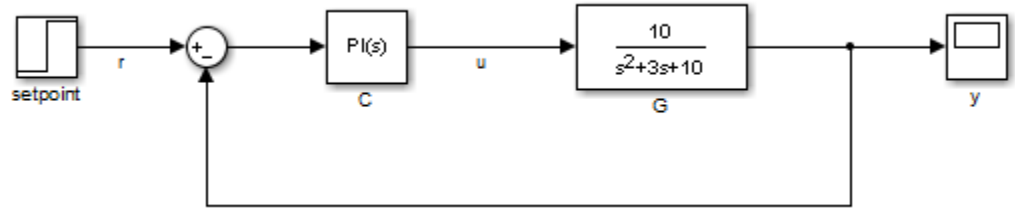
```
Tuy = getIOTransfer(T,'u','y');
stepplot(Tuy)
```

Analysis points are also useful to specify design requirements when tuning control systems with the `systune` command. For example, you can create a requirement that attenuates disturbances at the plant input by a factor of 10 (20 dB) or more.

```
Req = TuningGoal.Rejection('u',10);
```

Specify Analysis Points for MATLAB Models

Consider an LTI model of the following block diagram.



```
G = tf(10,[1 3 10]);
C = pid(0.2,1.5);
T = feedback(G*C,1);
```

With this model, you can obtain the closed-loop response from `r` to `y`. However, you cannot analyze the open-loop response at the plant input or simulate the rejection of a step disturbance at the plant input. To enable such analysis, mark the signal `u` as an analysis point by inserting an `AnalysisPoint` block between the plant and controller.

```
AP = AnalysisPoint('u');
T = feedback(G*AP*C,1);
T.OutputName = 'y';
```

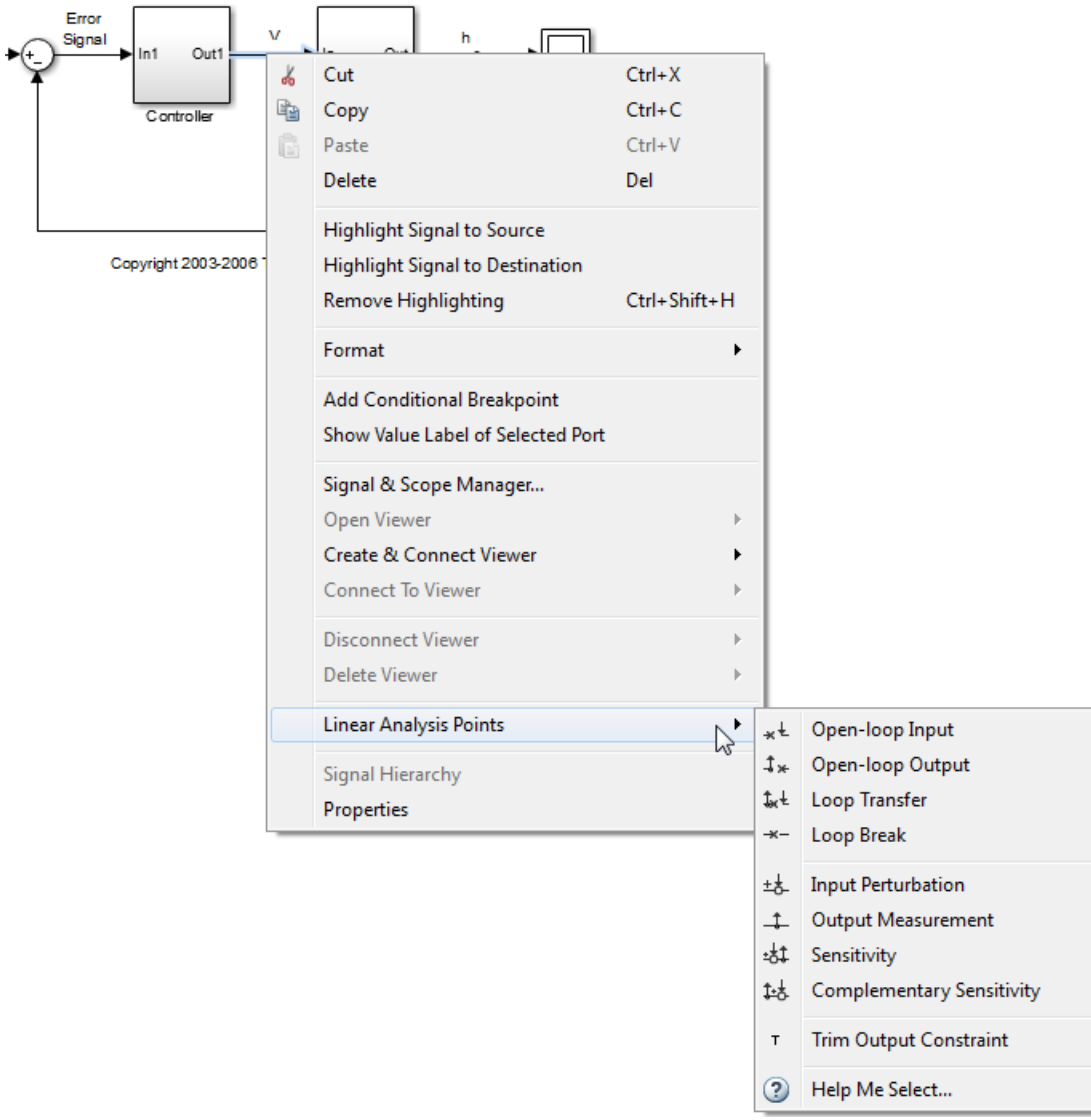
The plant input, `u`, is now available for analysis.

In creating the model `T`, you manually created the analysis point block `AP` and explicitly included it in the feedback loop. When you combine models using the `connect` command, you can instruct the software to insert analysis points automatically at the locations you specify. For more information, see `connect`.

Specify Analysis Points for Simulink Models

In Simulink, you can mark analysis points either explicitly in the block diagram, or programmatically using the `addPoint` command for `slLinearizer` or `slTuner` interfaces.

To mark an analysis point explicitly in the model, right-click a signal and, under **Linear Analysis Points**, select an analysis point type.



You can select any of the following closed-loop analysis point types, which are equivalent within an `slLinearizer` or `slTuner` interface; that is, they are treated the same way

by analysis functions, such as `getIOTransfer`, and tuning goals, such as `TuningGoal.StepTracking`.

- **Input Perturbation**
- **Output Measurement**
- **Sensitivity**
- **Complementary Sensitivity**

If you want to introduce a permanent loop opening at a signal as well, select one of the following open-loop analysis point types:

- **Open-Loop Input**
- **Open-Loop Output**
- **Loop Transfer**
- **Loop Break**

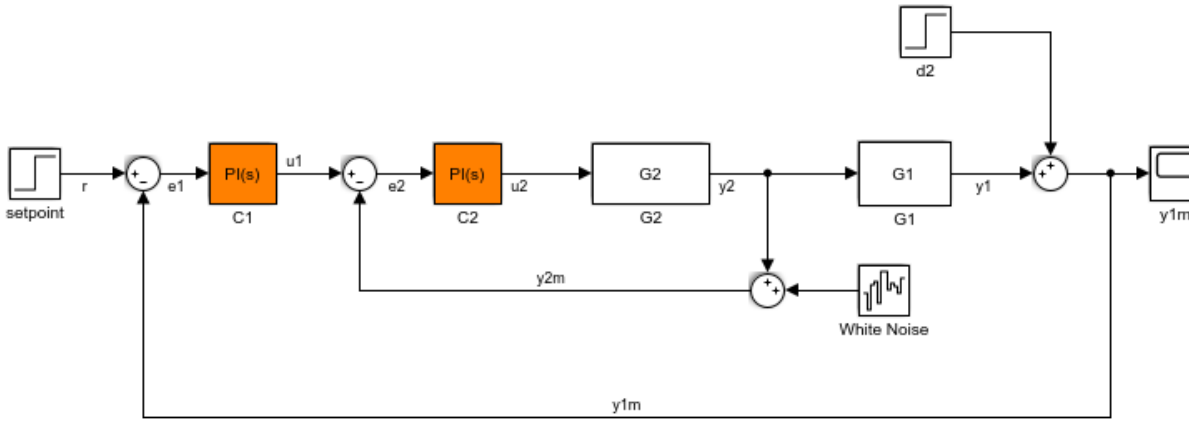
When you define a signal as an open-loop point, analysis functions such as `getIOTransfer` always enforce a loop break at that signal during linearization. All open-loop analysis point types are equivalent within an `slLinearizer` or `slTuner` interface. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-39.

When you create an `slLinearizer` or `slTuner` interface for a model, any analysis points defined in the model are automatically added to the interface. If you defined an analysis point using:

- A closed-loop type, the signal is added as an analysis point only.
- An open-loop type, the signal is added as both an analysis point and a permanent opening.

To mark analysis points programmatically, use the `addPoint` command. For example, consider the `scdcascade` model.

```
open_system('scdcascade')
```



To mark analysis points, first create an `slTuner` interface.

```
ST = slTuner('scdcascade');
```

To add a signal as an analysis point, use the `addPoint` command, specifying the source block and port number for the signal.

```
addPoint(ST,'scdcascade/C1',1);
```

If the source block has a single output port, you can omit the port number.

```
addPoint(ST,'scdcascade/G2');
```

For convenience, you can also mark analysis points using the:

- Name of the signal.

  ```
  addPoint(ST,'y2');
  ```

- Combined source block path and port number.

  ```
  addPoint(ST,'scdcascade/C1/1')
  ```

- End of the full source block path when unambiguous.

  ```
  addPoint(ST,'G1/1')
  ```

You can also add permanent openings to an `slLinearizer` or `slTuner` interface using the `addOpening` command, and specifying signals in the same way as for `addPoint`. For

more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-39.

```
addOpening(ST,'y1m');
```

You can also define analysis points by creating linearization I/O objects using the `linio` command.

```
io(1) = linio('scdcascade/C1',1,'input');
io(2) = linio('scdcascade/G1',1,'output');
addPoint(ST,io);
```

As when you define analysis points directly in your model, if you specify a linearization I/O object with:

- A closed-loop type, the signal is added as an analysis point only.
- An open-loop type, the signal is added as both an analysis point and a permanent opening.

When you specify response I/Os in a tool such as Linear Analysis Tool or **Control System Tuner**, the software creates analysis points as needed.

Refer to Analysis Points for Analysis and Tuning

Once you have marked analysis points, you can analyze the response at any of these points using the following analysis functions:

- `getIOTransfer` — Transfer function for specified inputs and outputs
- `getLoopTransfer` — Open-loop transfer function from an additive input at a specified point to a measurement at the same point
- `getSensitivity` — Sensitivity function at a specified point
- `getCompSensitivity` — Complementary sensitivity function at a specified point

You can also create tuning goals that constrain the system response at these points. The tools to perform these operations operate in a similar manner for models created at the command line and models created in Simulink.

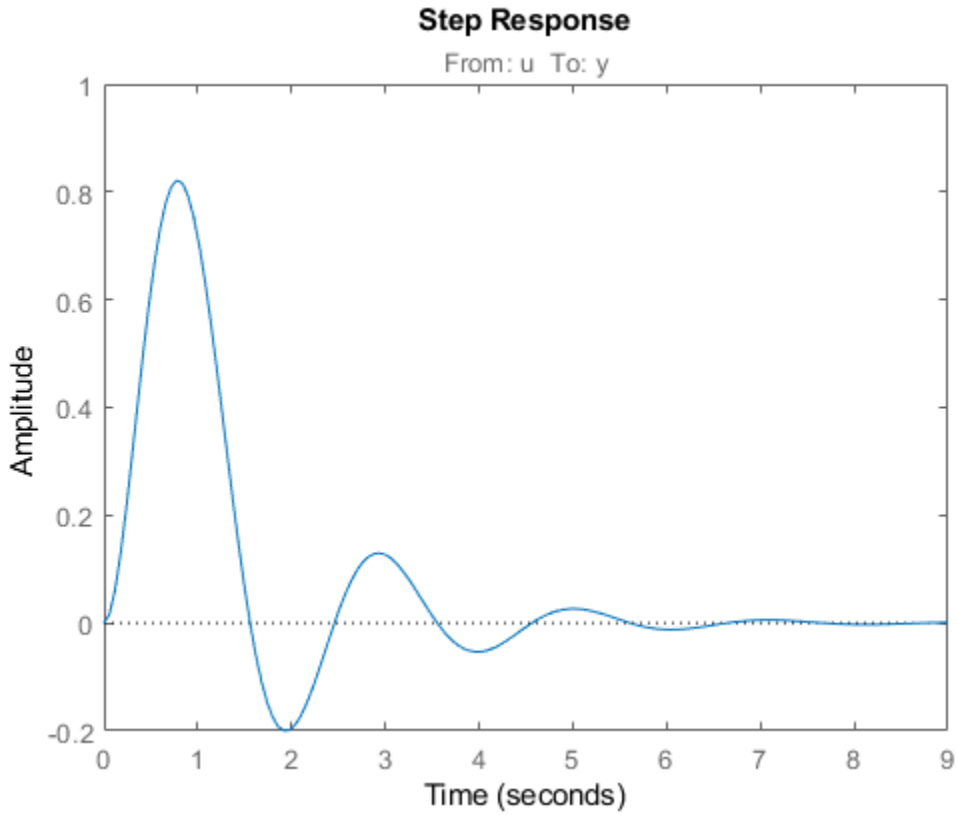
To view the available analysis points, use the `getPoints` function. You can view the analysis for models created:

- At the command line:
- In Simulink:

For closed-loop models created at the command line, you can also use the model input and output names when:

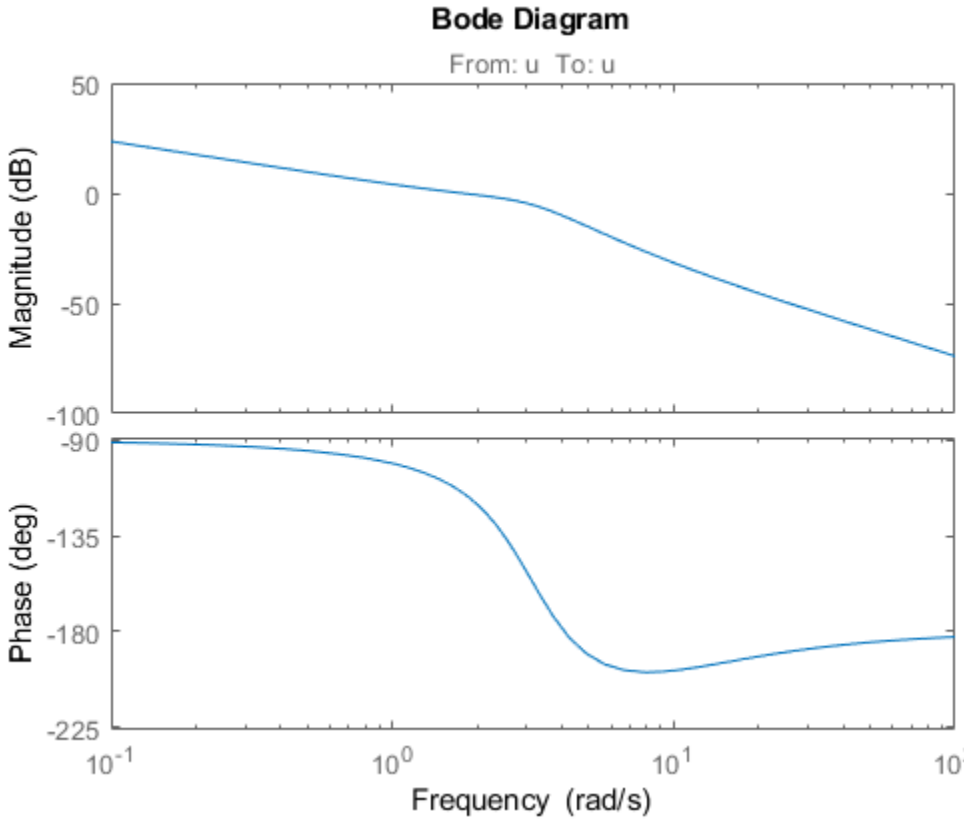
- Computing a closed-loop response.

```
ioSys = getIOTransfer(T,'u','y');
stepplot(ioSys)
```



- Computing an open-loop response.

```
loopSys = getLoopTransfer(T,'u',-1);
bodeplot(loopSys)
```



- Creating tuning goals for `systune`.

```
R = TuningGoal.Margins('u',10,60);
```

Use the same method to refer to analysis points for models created in Simulink. In Simulink models, for convenience, you can use any unambiguous abbreviation of the analysis point names returned by `getPoints`.

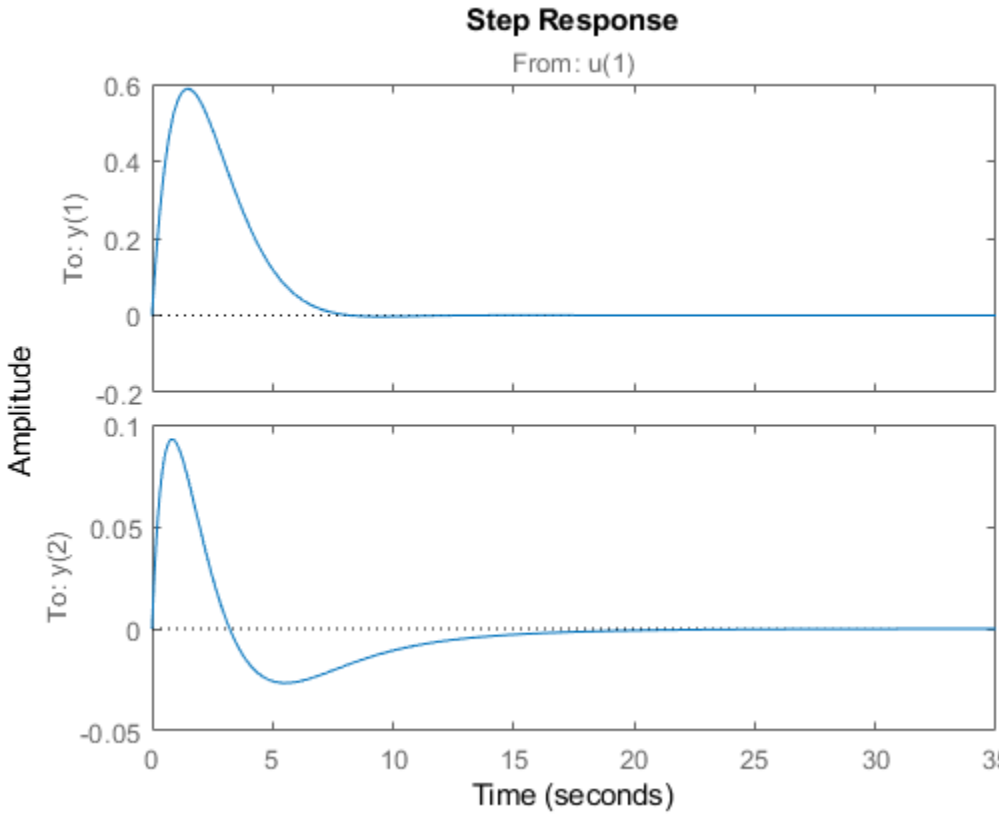
```
ioSys = getIOTransfer(ST,'u1','y1');
sensG2 = getSensitivity(ST,'G2');
R = TuningGoal.Margins('u1',10,60);
```

Finally, if some analysis points are vector-valued signals or multichannel locations, you can use indices to select particular entries or channels. For example, suppose `u` is a two-entry vector in a closed-loop MIMO model.

```
G = ss([-1 0.2;0 -2],[1 0;0.3 1],eye(2),0);
C = pid(0.2,0.5);
AP = AnalysisPoint('u',2);
T = feedback(G*AP*C,eye(2));
T.OutputName = 'y';
```

You can compute the open-loop response of the second channel and measure the impact of a disturbance on the first channel.

```
L = getLoopTransfer(T,'u(2)',-1);
stepplot(getIOTransfer(T,'u(1)','y'))
```



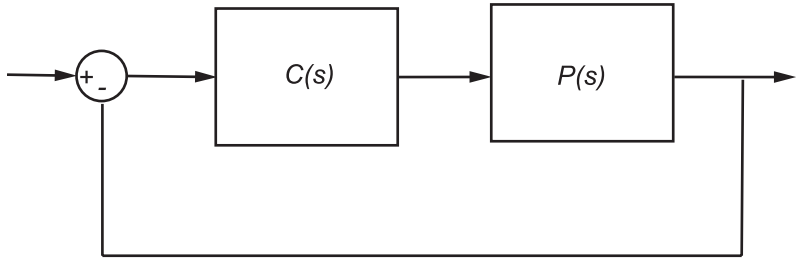
When you create tuning goals in **Control System Tuner**, the software creates analysis points as needed.

See Also

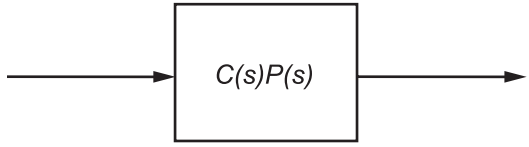
`addPoint` | `getIOTransfer` | `getPoints` | `slLinearizer`

Compute Open-Loop Response

The open-loop response of a control system is the combined response of the plant and the controller, excluding the effect of the feedback loop. For example, the following block diagram shows a single-loop control system.

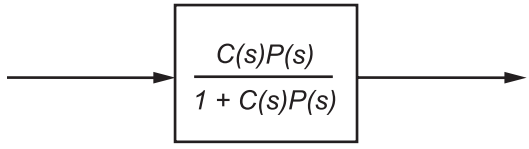


If the controller, $C(s)$, and plant, $P(s)$, are linear, the corresponding open-loop transfer function is $C(s)P(s)$.

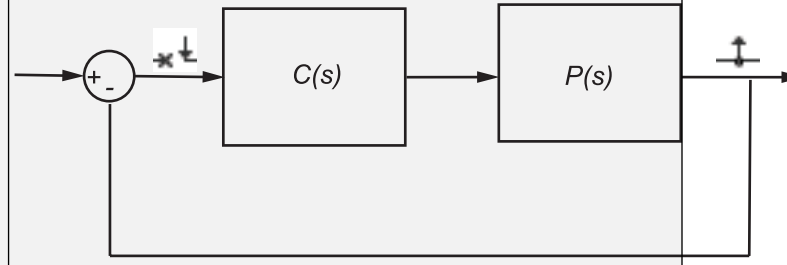
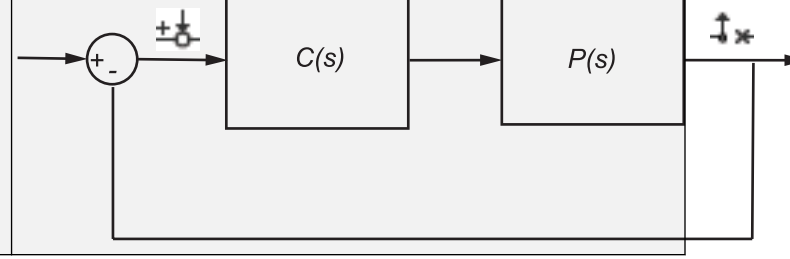


To remove the effects of the feedback loop, insert a loop opening analysis point without manually breaking the signal line. Manually removing the feedback signal from a nonlinear model changes the model operating point and produces a different linearized model. For more information, see “How the Software Treats Loop Openings” on page 2-39.

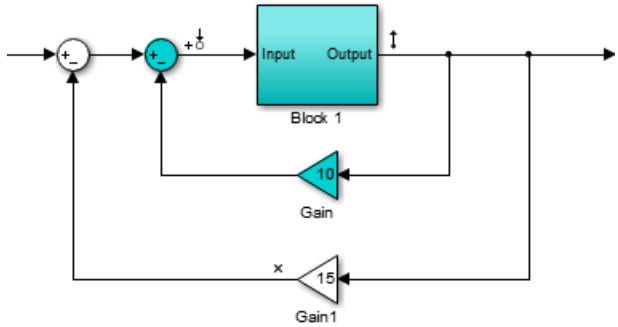
If you do not insert a loop opening, the resulting linear model includes the effects of the feedback loop.



To specify the loop opening for this example, you can use either of the following analysis points.

Analysis Point	Description	To compute $C(s)P(s)$
Open-loop input	Specifies a loop opening followed by an input perturbation.	Specify an open-loop input at the input to the controller and an output measurement at the output of the plant. 
Open-loop output	Specifies an output measurement followed by a loop break.	Specify an open-loop output at the output of the plant and an input perturbation at the input of the controller. 

For some systems, you cannot specify the loop opening at the same location as the linearization input or output point. For example, to open the outer loop in the following system, a loop opening point is added to the feedback path using a loop break analysis point . As a result, only the blue blocks are on the linearization path.



Placing the loop opening at the same location as the input or output signal would also remove the effect of the inner loop from the linearization result.

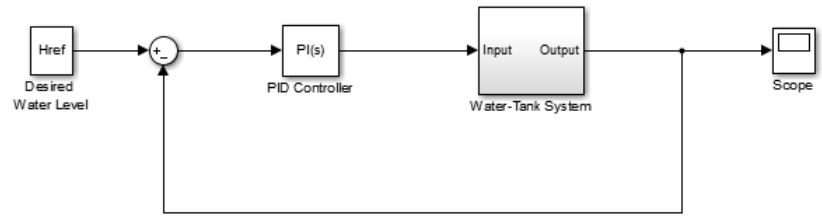
You can specify analysis points directly in your Simulink model, in the Linear Analysis Tool, or at the command line. For more information, about the different types of analysis points and how to define them, see “Specify Portion of Model to Linearize” on page 2-13.

Compute Open-Loop Response Using Linear Analysis Tool

This example shows how to compute a linear model of the combined controller-plant system without the effects of the feedback signal. You can analyze the resulting linear model using, for example, a Bode plot.

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```

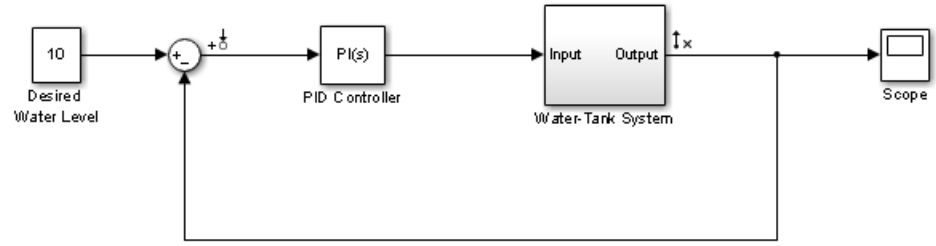


The Water-Tank System block represents the plant in this control system and contains all of the system nonlinearities.

In the Simulink Editor, specify the portion of the model to linearize. For this example, specify the loop opening using open-loop output analysis point.

1. Right-click the PID Controller block input signal (the output of the Sum block), and select **Linear Analysis Points > Input Perturbation**.
2. Right-click the Water-Tank System output signal, and select **Linear Analysis Points > Open-loop Output**.

Annotations appear in the model indicating which signals are designated as analysis points.

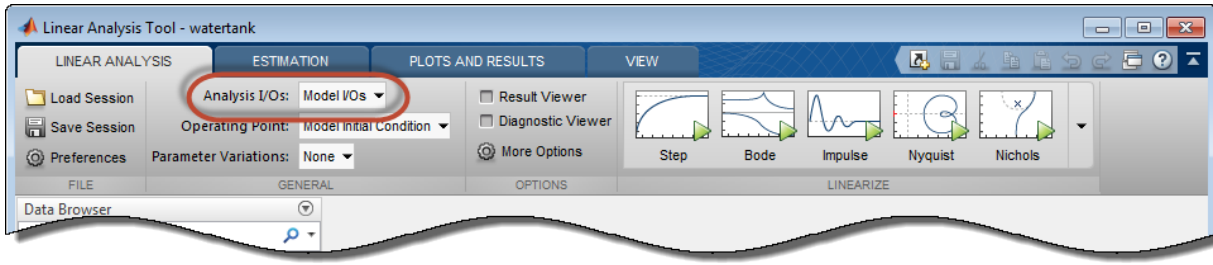


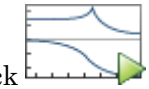
Tip If you do not want to introduce changes to the Simulink model, you can specify the analysis points in the Linear Analysis Tool. For more information, see "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29.

Open the Linear Analysis Tool for the model.

In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

By default, the analysis points you specified in the model are selected for linearization, as displayed in the **Analysis I/Os** drop-down list.

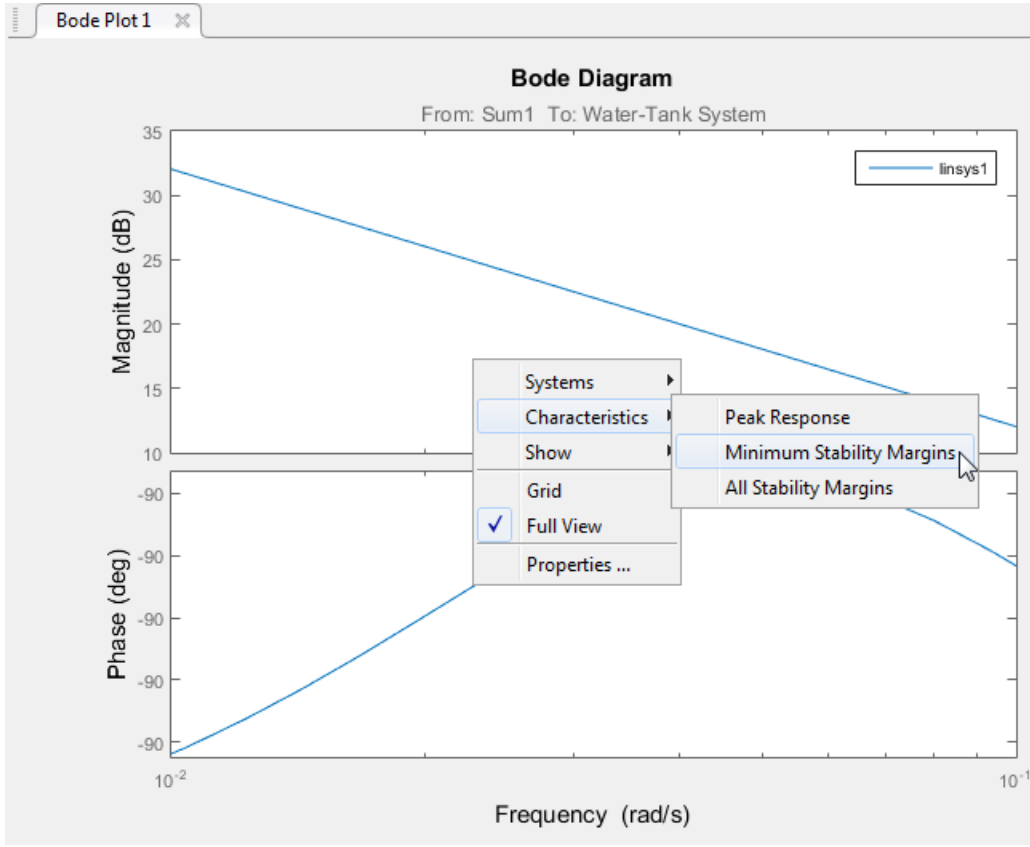


To linearize the model using the specified analysis points and generate a Bode plot of the linearized model, click  **Bode**.

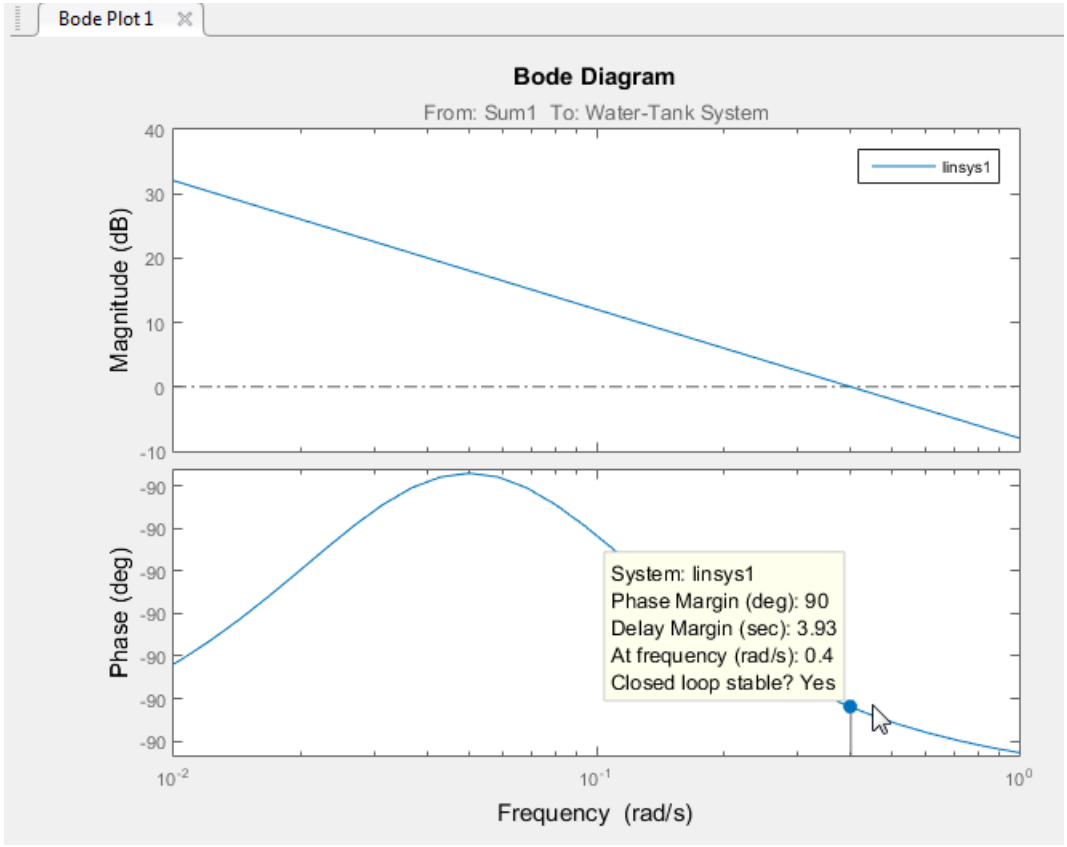
By default, the Linear Analysis Tool linearizes the model at the model initial conditions, as shown in the **Operating Point** drop-down list. For examples of linearizing a model at a different operating point, see “Linearize at Trimmed Operating Point” on page 2-85 and “Linearize at Simulation Snapshot” on page 2-91.

Tip To generate response types other than a Bode plot, click the corresponding button in the plot gallery.

To view the minimum stability margins for the model, right-click the Bode plot, and select **Characteristics > Minimum Stability Margins**.



The Bode plot displays the phase margin marker. To show a data tip that contains the phase margin value, click the marker.



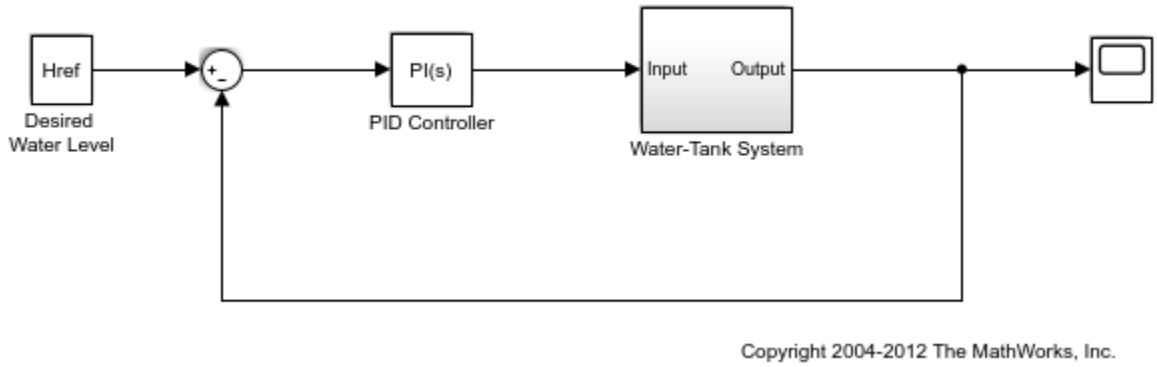
For this system, the phase margin is 90 degrees at a crossover frequency of 0.4 rad/s.

Compute Open-Loop Response at the Command Line

This example shows how to compute a linear model of the combined controller-plant system without the effects of the feedback signal. You can analyze the resulting linear model using, for example, a Bode plot.

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Specify the portion of the model to linearize by creating an array of analysis points using the `linio` command:

- Open-loop input point at the input of the PID Controller block. This signal originates at the output of the Sum1 block.
- Output measurement at the output of the Water-Tank System block.

```
io(1) = linio('watertank/Sum1',1,'openinput');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

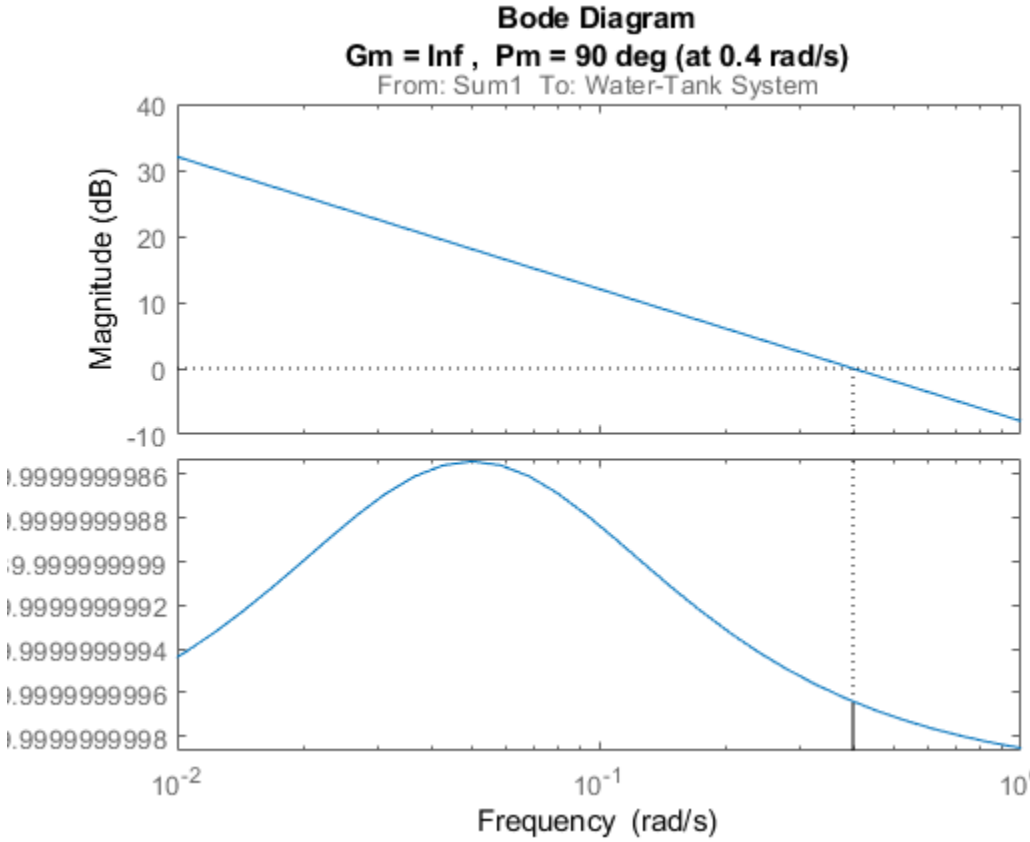
The open-loop input analysis point includes a loop opening, which breaks the signal flow and removes the effects of the feedback loop.

Linearize the model at the default model operating point using the `linearize` command.

```
linsys = linearize(sys,io);
```

`linsys` is the linearized open-loop transfer function of the system. You can now analyze the response by, for example, plotting its frequency response and viewing the gain and phase margins.

```
margin(linsys)
```



For this system, the gain margin is infinite, and the phase margin is 90 degrees at a crossover frequency of 0.4 rad/s.

See Also

Linear Analysis Tool | `linearize`

More About

- "Specify Portion of Model to Linearize" on page 2-13
- "How the Software Treats Loop Openings" on page 2-39

- "Linearize Simulink Model at Model Operating Point" on page 2-69
- "Linearize Plant" on page 2-41
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77

Linearize Simulink Model at Model Operating Point

When linearizing a Simulink model, if you do not specify an operating point, the software uses the operating point specified in the model by default. The model operating point consists of the initial state and input signal values stored in the model.

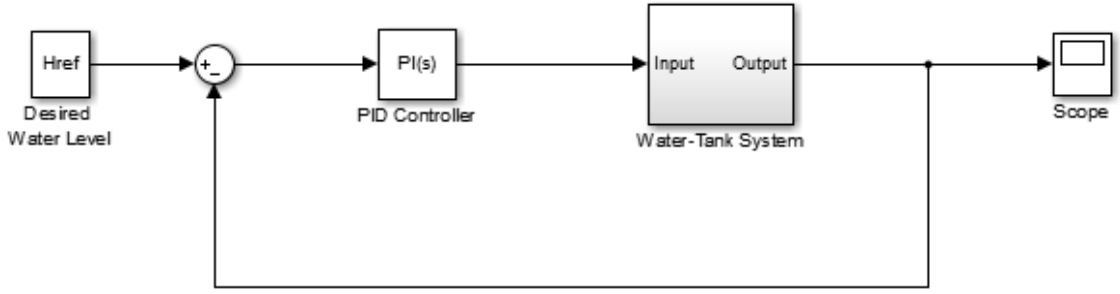
For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-85 and “Linearize at Simulation Snapshot” on page 2-91.

Linearize Simulink Model Using Linear Analysis Tool

This example shows how to linearize a Simulink model at the operating point specified in the model using the **Linear Analysis Tool**.

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



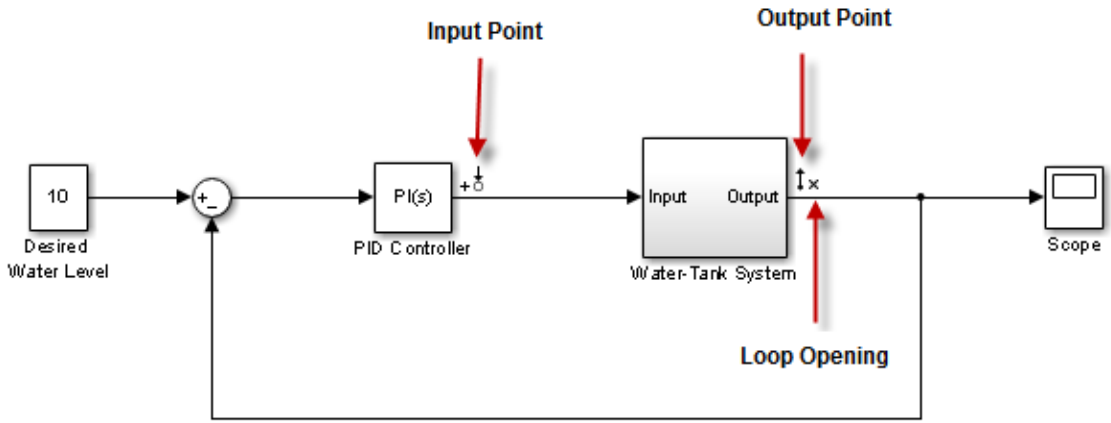
The Water-Tank System block represents the plant in this control system and includes all of the system nonlinearities.

In the Simulink window, specify the portion of the model to linearize:

1 To specify the linearization input, right-click the output signal of the PID Controller block, and select **Linear Analysis Points** >  **Input Perturbation**.

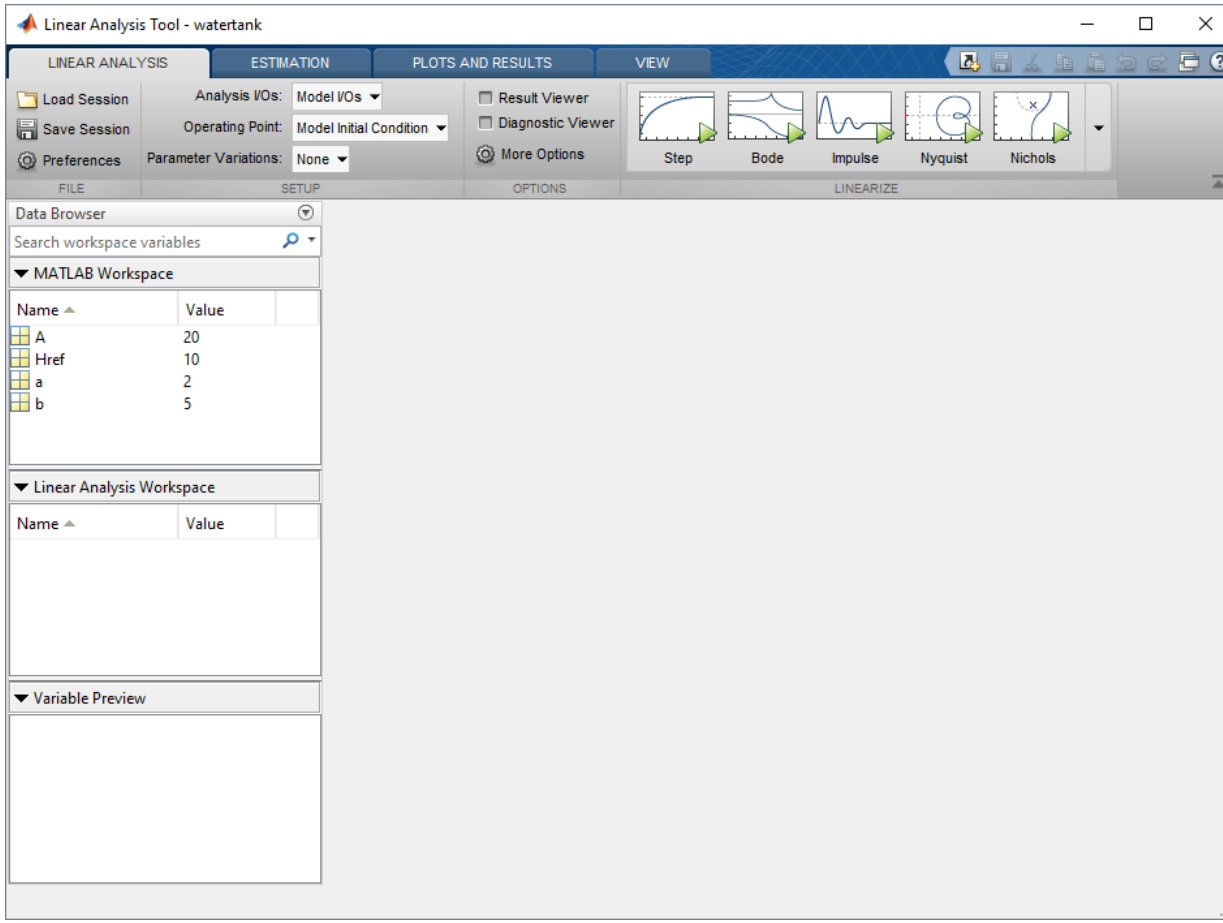
2 To specify the linearization output, right-click the output signal of the Water-Tank System, and select **Linear Analysis Points** > **Open-loop Output**. An open-loop output point is an output measurement followed by a loop opening, which removes the effects of the feedback signal on the linearization without changing the model operating point.

When you add linear analysis points, the software adds markers at their respective locations in the model. For more information on the different types of analysis points, see "Specify Portion of Model to Linearize" on page 2-13.



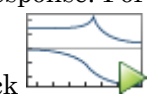
For more information on defining analysis points in a Simulink model, see "Specify Portion of Model to Linearize in Simulink Model" on page 2-21. Alternatively, if you do not want to introduce changes to the Simulink model, you can define analysis points using the Linear Analysis Tool. For more information, see "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29.

To open the Linear Analysis Tool for the model, in the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

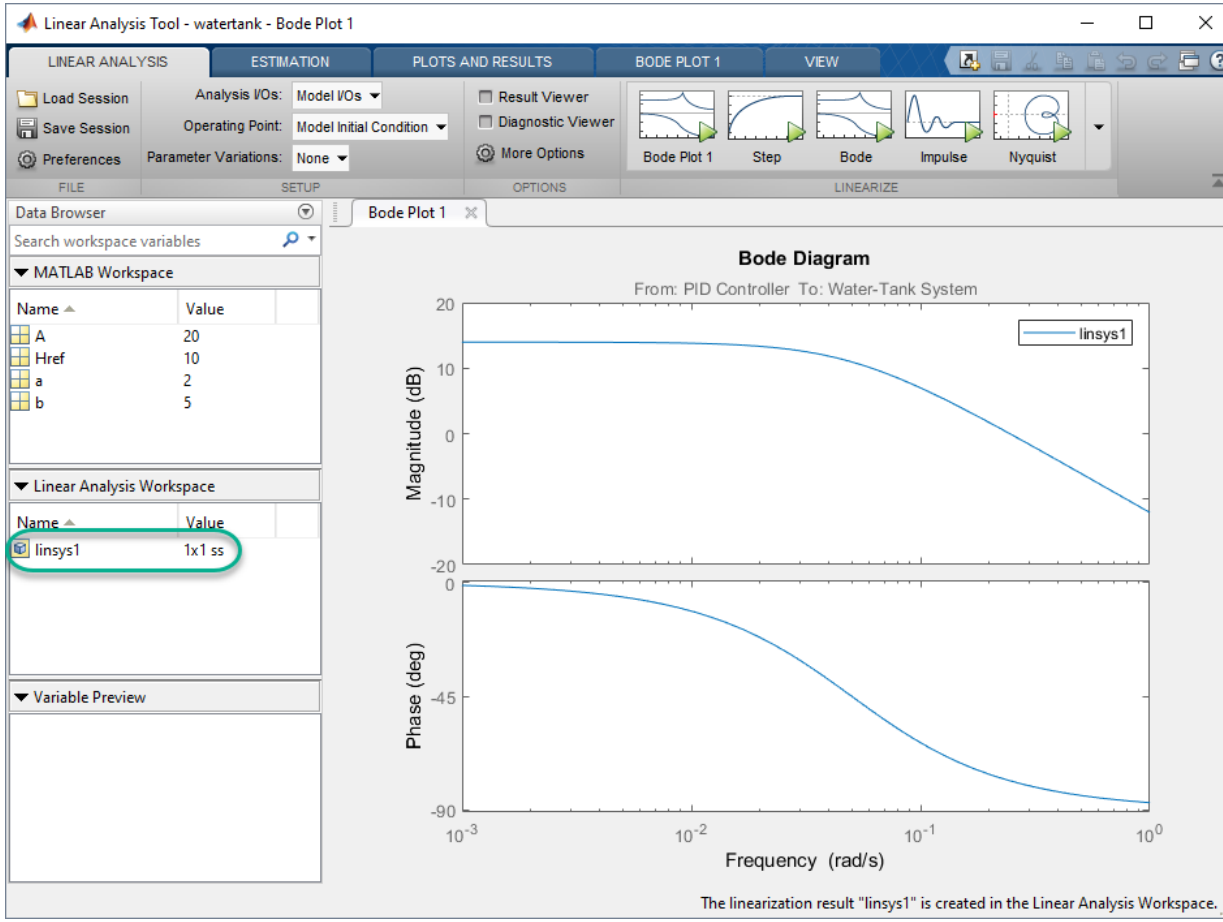


To use the analysis points you defined in the Simulink model as linearization I/Os, on the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, leave `Model I/Os` selected.

For this example, use the model operating point for linearization. In the **Operating Point** drop-down list, leave `Model Initial Condition` selected.

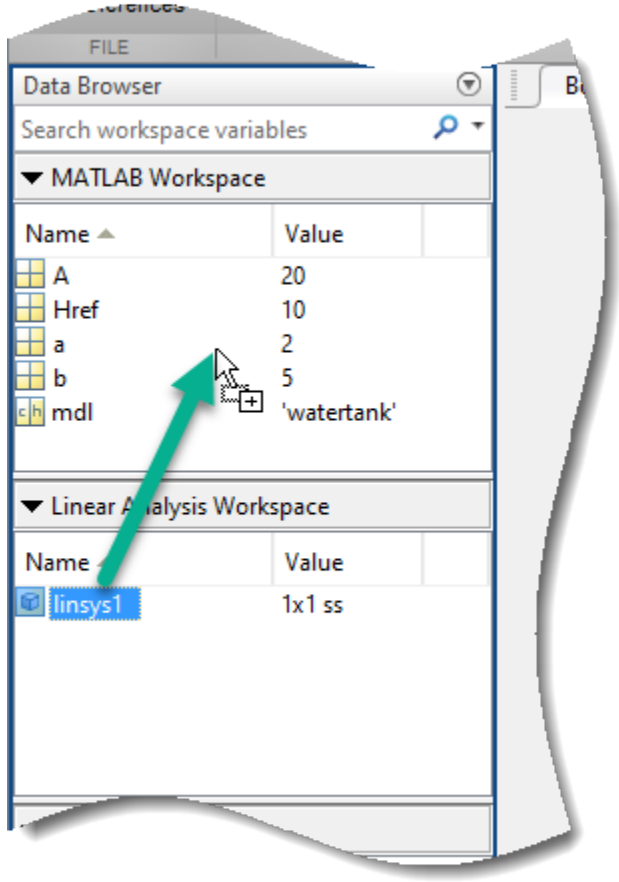
To linearize the system and generate a response plot for analysis, in the **Linearize** section, click a response. For this example, to generate a Bode plot for the resulting linear model, click  **Bode**.

The software adds the linearized model, `linsys1`, to the **Linear Analysis Workspace** and generates a Bode plot for the model. `linsys1` is the linear model from the specified input to the specified output, computed at the default model operating point.



For more information on analyzing linear models, see “Analyze Results Using Linear Analysis Tool Response Plots” on page 2-146.

You can also export the linearized model to the MATLAB workspace. To do so, in the **Data Browser**, drag `linsys1` from the **Linear Analysis Workspace** to the **MATLAB Workspace**.

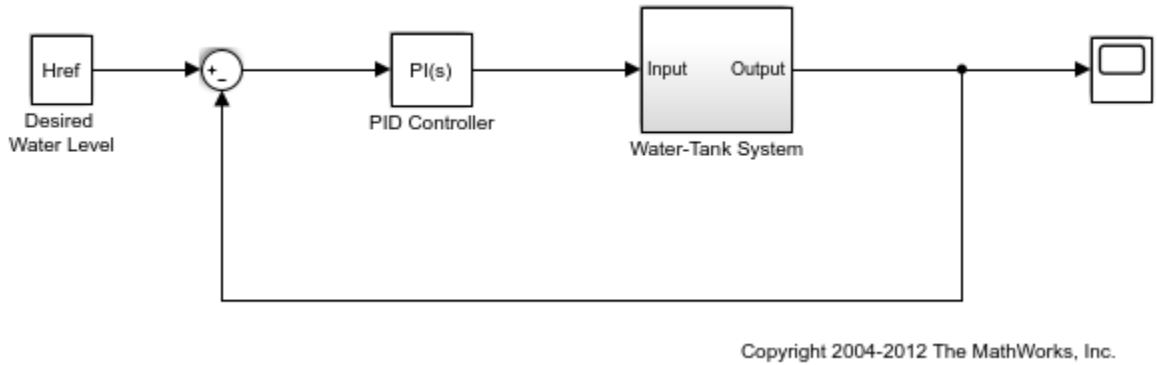


Linearize Simulink Model at Command Line

This example shows how to linearize a Simulink® model at the model operating point using the `linearize` command.

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



For this system, the Water-Tank System block contains all the nonlinear dynamics. To specify the portion of the model to linearize, create an array of linearization I/O objects using the `linio` command.

Create an input perturbation analysis point at the output of the PID Controller block.

```
io(1) = linio('watertank/PID Controller',1,'input');
```

Create an open-loop output analysis point at the output of the Water-Tank System block. An open-loop output point is an output measurement followed by a loop opening, which removes the effects of the feedback signal on the linearization without changing the model operating point.

```
io(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

For information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-13.

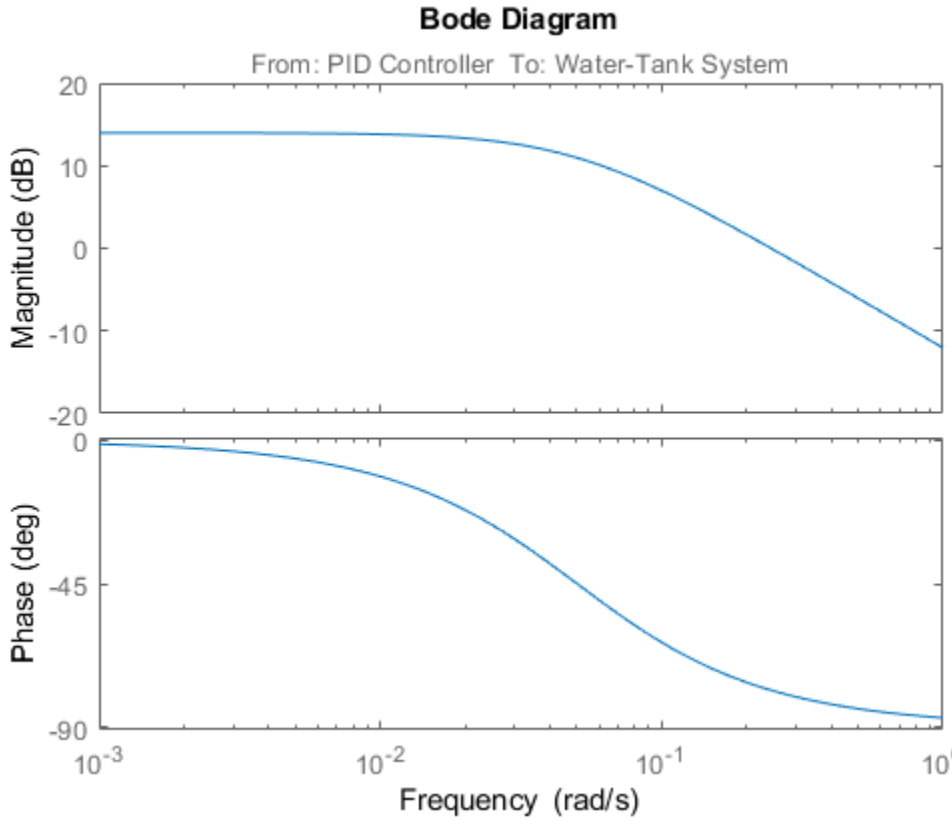
Linearize the model at the model operating point using the specified analysis points.

```
linsys1 = linearize(mdl,io);
```

`linsys1` is the linear model from the specified input to the specified output, computed at the default model operating point.

You can then analyze the response of the linearized model. For example, plot its Bode response.

```
bode(linsys1)
```



For more information on analyzing linear models, see “Linear Analysis” (Control System Toolbox).

See Also

Linear Analysis Tool | `linearize`

More About

- “Linearize at Trimmed Operating Point” on page 2-85
- “Linearize at Simulation Snapshot” on page 2-91

- "Linearize at Triggered Simulation Events" on page 2-95
- "Linearize Plant" on page 2-41
- "Compute Open-Loop Response" on page 2-59
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77

Visualize Bode Response of Simulink Model During Simulation

This example shows how to visualize linear system characteristics of a nonlinear Simulink model during simulation, computed at the model operating point (simulation snapshot time of 0).

1 Open Simulink model.

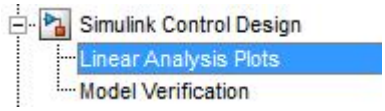
For example:

```
open_system('watertank')
```

2 Open the Simulink Library Browser by selecting **View > Library Browser** in the model window.

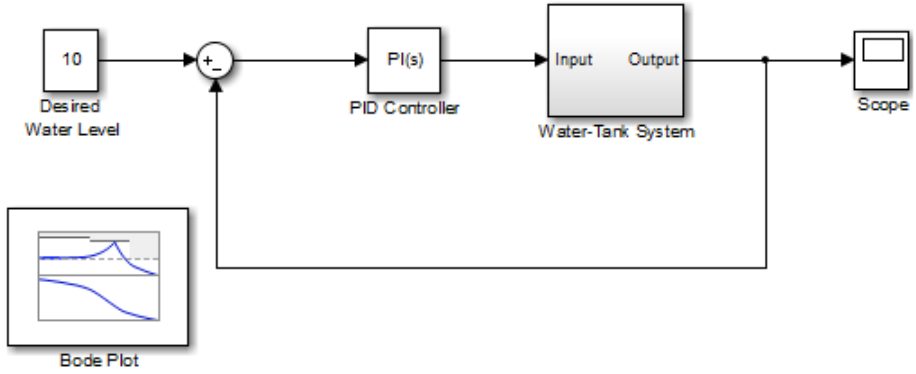
3 Add a plot block to the Simulink model.

a In the **Simulink Control Design** library, select **Linear Analysis Plots**.

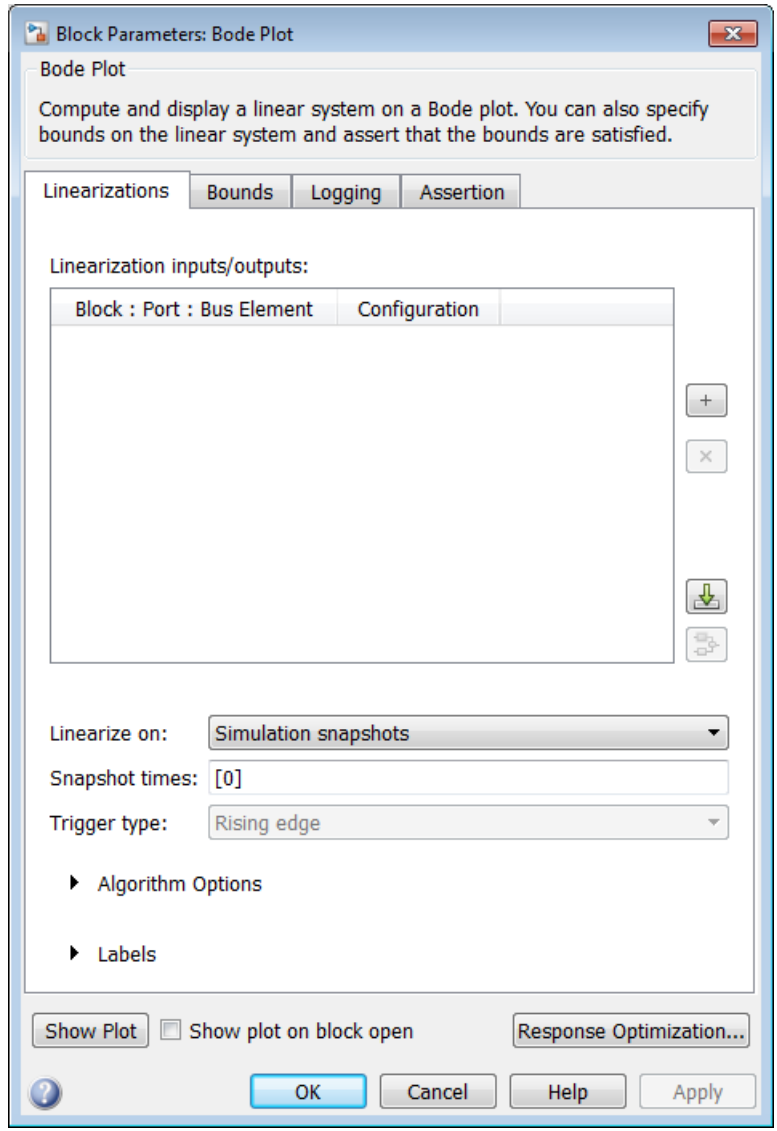


b Drag and drop a block, such as the Bode Plot block, into the model window.

The model now resembles the following figure.



4 Double-click the block to open the Block Parameters dialog box.



To learn more about the block parameters, see the block reference pages.

5 Specify the linearization I/O points.

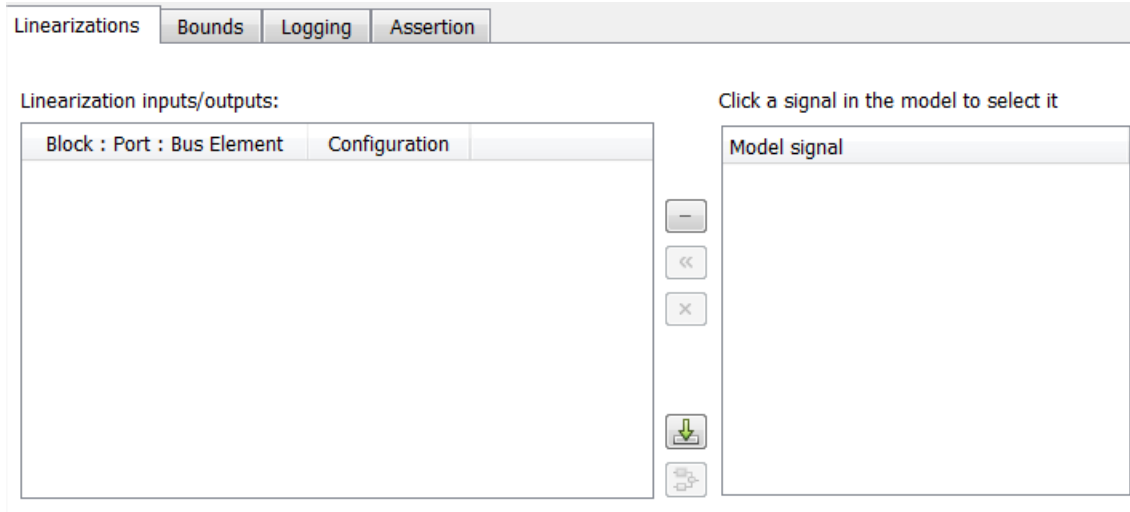
The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

a To specify an input:

i Click  adjacent to the **Linearization inputs/outputs** table.

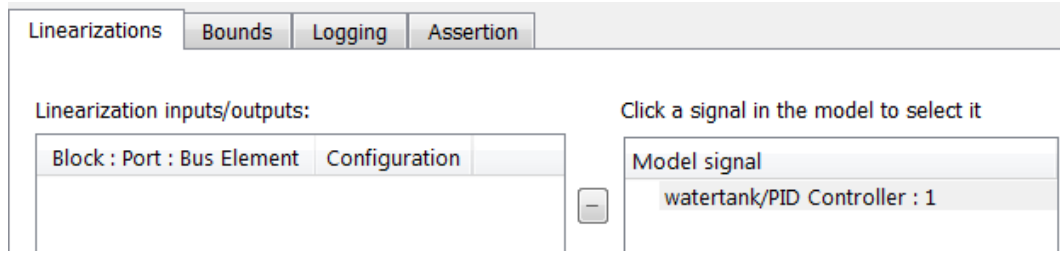
The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.



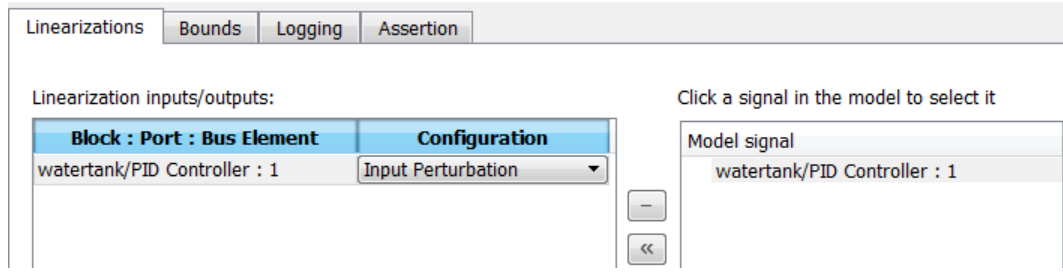
Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



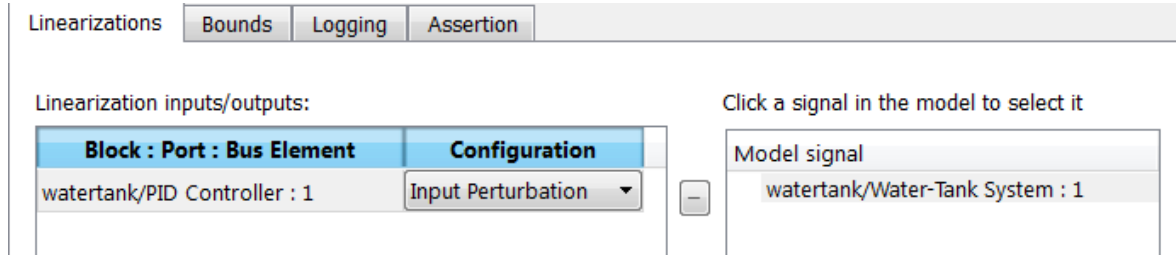
iii Click « to add the signal to the **Linearization inputs/outputs** table.



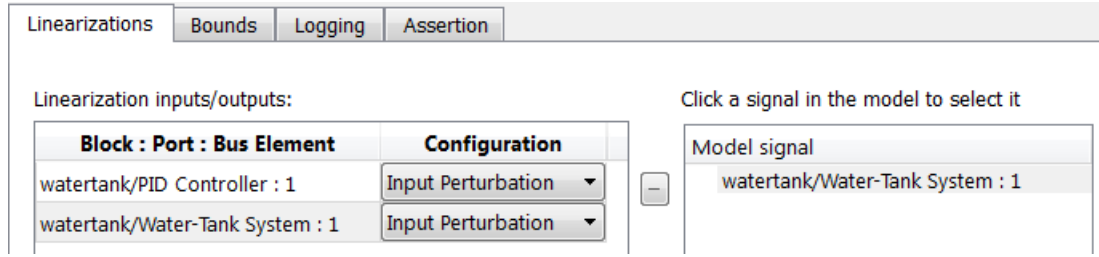
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.

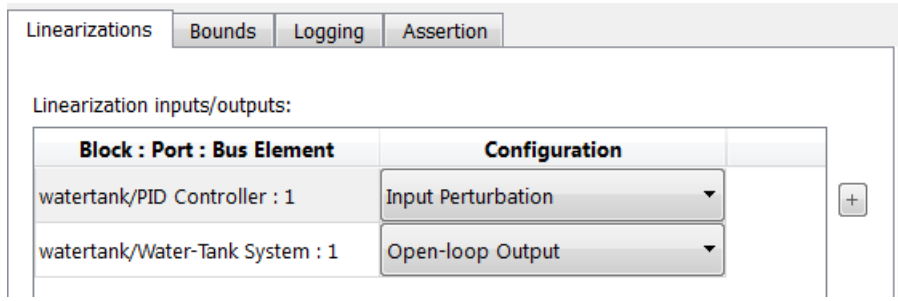


ii Click « to add the signal to the **Linearization inputs/outputs** table.



iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



c Click  to collapse the **Click a signal in the model to select it** area.

Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear in the **Linearization inputs/outputs** table.

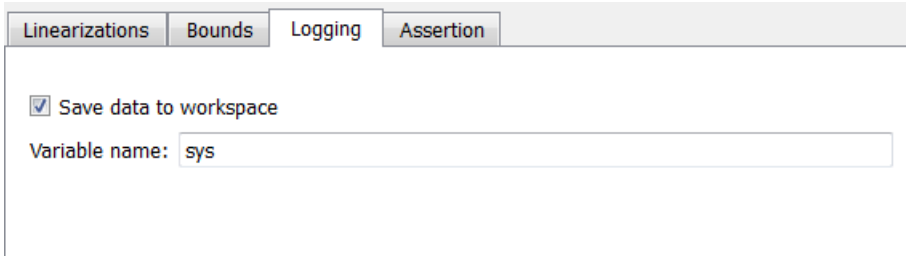
---

6 Save the linear system.

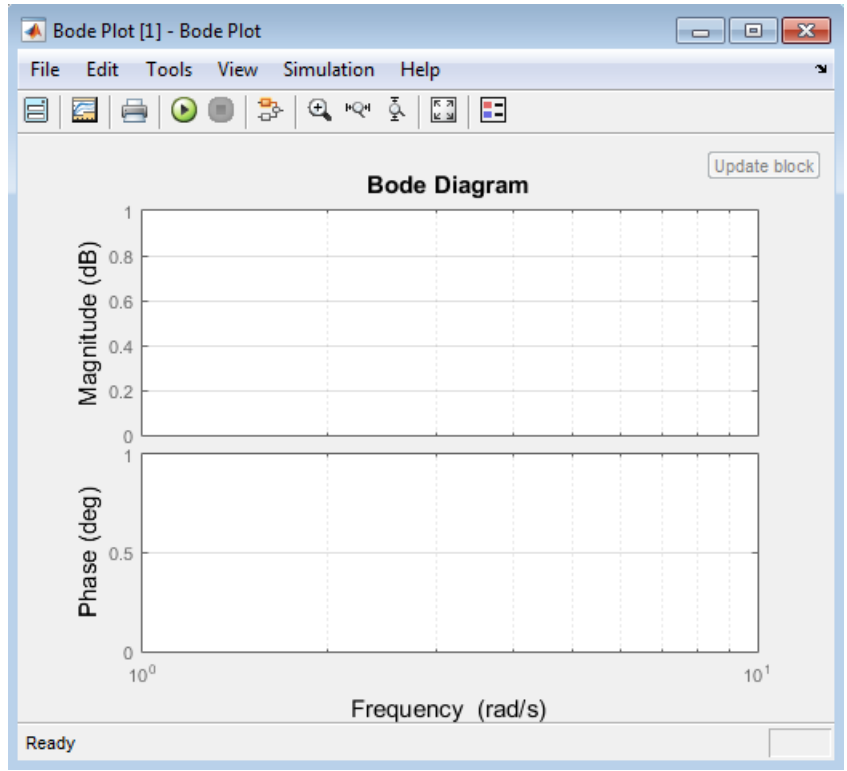
a Select the **Logging** tab.

b Select the **Save data to workspace** option, and specify a variable name in the **Variable name** field.

The **Logging** tab now resembles the following figure.



7 Click **Show Plot** to open an empty plot.

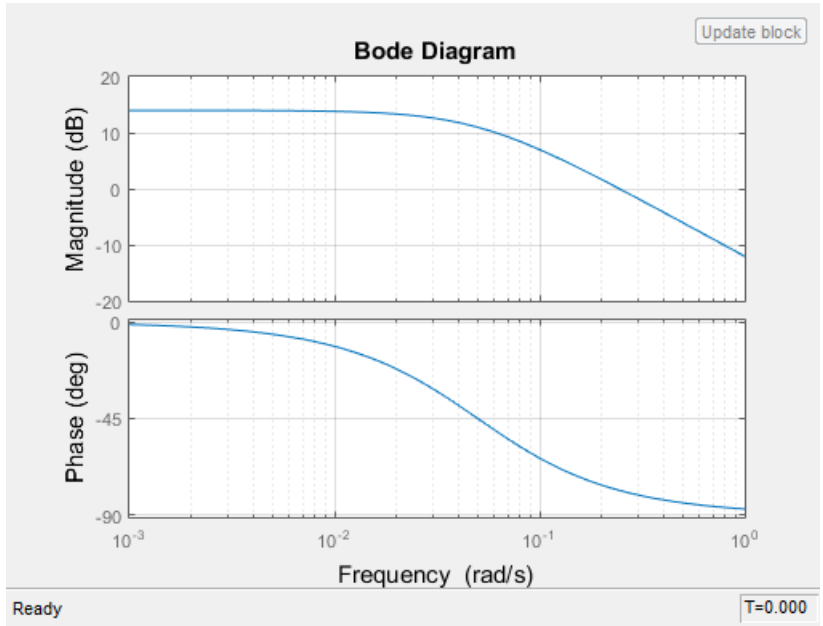


8 Plot the linear system characteristics by clicking  in the plot window.

Alternatively, you can simulate the model from the model window.

The software linearizes the portion of the model between the linearization input and output at the default simulation time of 0, specified in **Snapshot times** parameter in the Block Parameters dialog box, and plots the Bode magnitude and phase.

After the simulation completes, the plot window resembles the following figure.



The computed linear system is saved as `sys` in the MATLAB workspace. `sys` is a structure with `time` and `values` fields. To view the structure, type:

```
sys
```

This command returns the following results:

```
sys =

         time: 0
       values: [1x1 ss]
    blockName: 'watertank/Bode Plot'
```

- The `time` field contains the default simulation time at which the linear system is computed.

- The `values` field is a state-space object which stores the linear system computed at simulation time of 0. To learn more about the properties of state-space objects, see `ss` in the Control System Toolbox documentation.

(If the Simulink model is configured to save simulation output as a single object, the data structure `sys` is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.)

See Also

Bode Plot

More About

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-110
- "Plotting Linear System Characteristics of a Chemical Reactor" on page 2-121
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-117

Linearize at Trimmed Operating Point

This example shows how to linearize a model at a trimmed steady-state operating point (equilibrium operating point) using the Linear Analysis Tool.

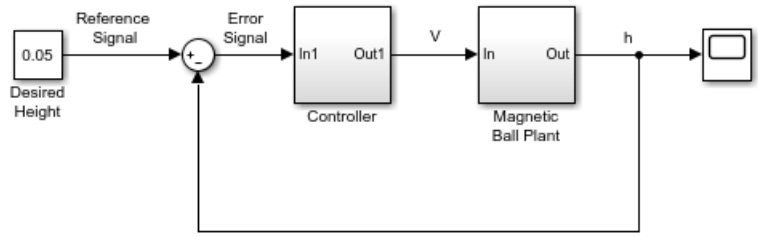
The operating point is *trimmed* by specifying constraints on the operating point values, and performing an optimization search that meets these state and input value specifications.

Code Alternative

Use `linearize`. For examples and additional information, see the `linearize` reference page.

1 Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```



2 Open the Linear Analysis Tool for the model.

In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

3 In the Simulink model window, define the portion of the model to linearize for this linearization task:

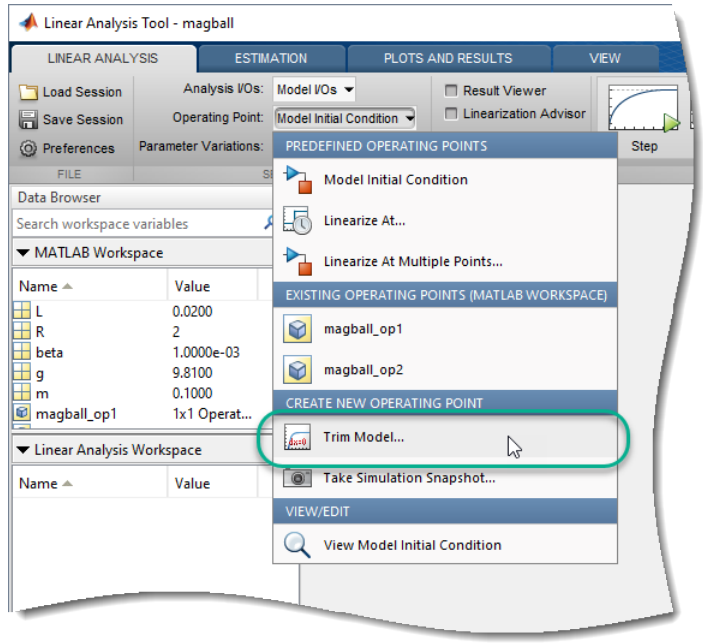
a Right-click the Controller block output signal (input signal to the plant). Select **Linear Analysis Points > Input Perturbation**.

b Right-click the Magnetic Ball Plant output signal, and select **Linear Analysis Points > Open-loop Output**.

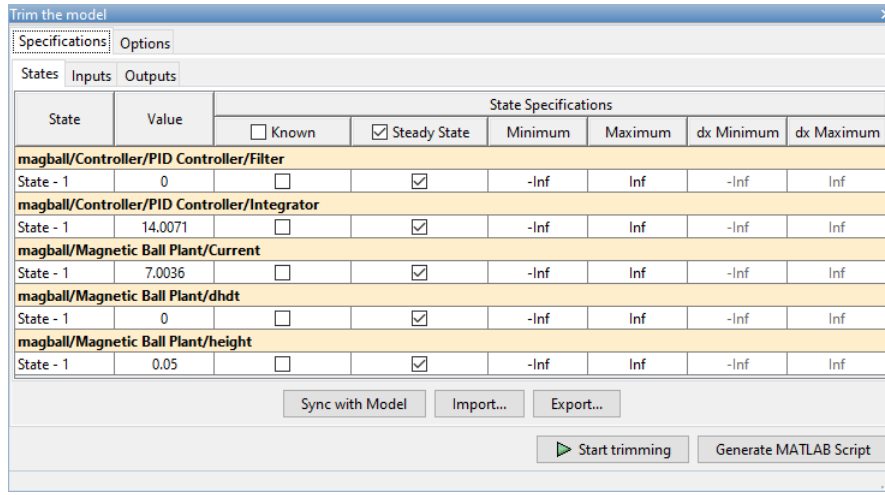
Annotations appear in the model indicating which signals are designated as analysis points.

Tip Alternatively, if you do not want to introduce changes to the Simulink model, you can specify the analysis points in the Linear Analysis Tool. For more information, see "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29.

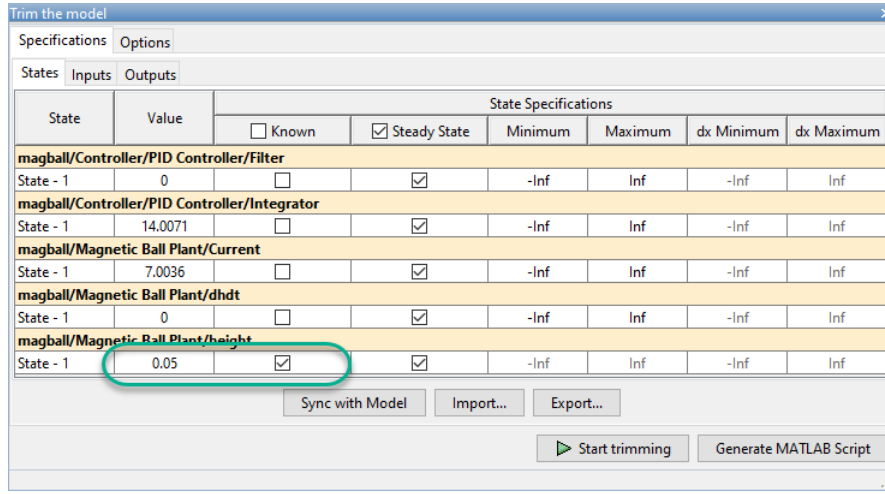
4 Create a new steady-state operating point at which to linearize the model. In the Linear Analysis Tool, in the **Operating Point** drop-down list, select **Trim model**.



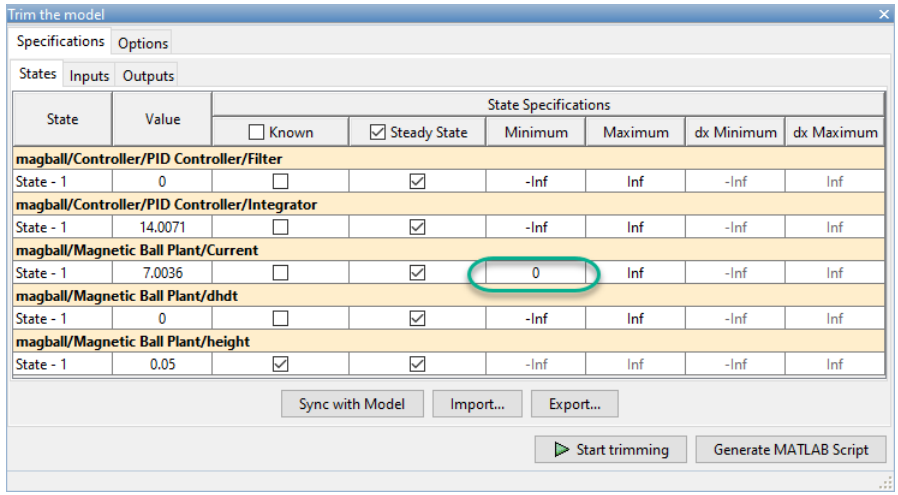
In the Trim the model dialog box, the **Specifications** tab shows the default specifications for model trimming. By default, all model states are specified to be at equilibrium, indicated by the check marks in the **Steady State** column.



5 Specify a steady-state operating point at which the magnetic ball height remains fixed at the reference signal value, 0.05. In the **States** tab, select **Known** for the **height** state. This selection tells Linear Analysis Tool to find an operating point at which this state value is fixed.



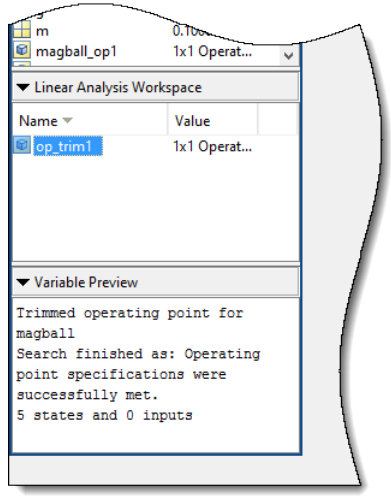
6 Since the ball height is greater than zero, the current must also be greater than zero. Enter `0` for the minimum bound of the **Current** block state.



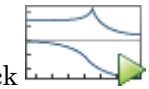
7 Compute the operating point.

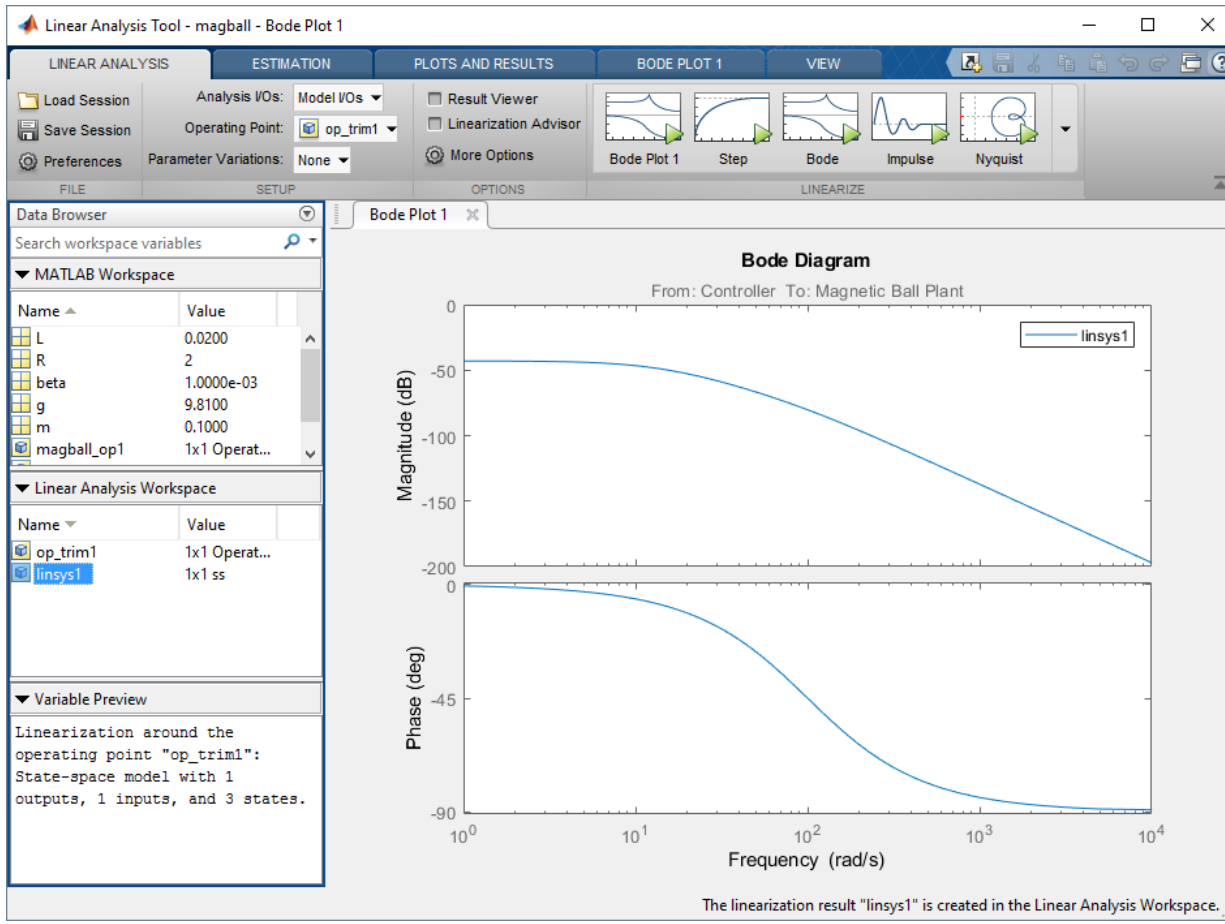
Click  **Start trimming**.

A new variable, `op_trim1`, appears in the Linear Analysis Workspace.



In the **Operating Point** drop-down list, this operating point is now selected as the operating point to be used for linearization.

8 Linearize the model at the specified operating point and generate a bode plot of the result. Click  **Bode**. The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the Linear Analysis Workspace.



Tip Instead of a Bode plot, generate other response types by clicking the corresponding button in the plot gallery.

Right-click on the plot and select information from the **Characteristics** menu to examine characteristics of the linearized response.

See Also

More About

- "Compute Steady-State Operating Point from State Specifications" on page 1-14
- "Compute Steady-State Operating Point from Output Specifications" on page 1-28

Linearize at Simulation Snapshot

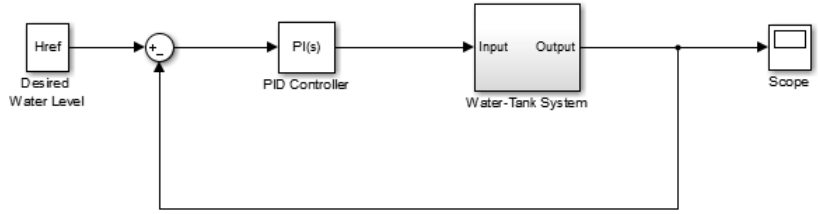
This example shows how to use the Linear Analysis Tool to linearize a model by simulating the model and extracting the state and input levels of the system at specified simulation times.

Code Alternative

Use `linearize`. For examples and additional information, see the `linearize` reference page.

1 Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



2 Open the Linear Analysis Tool for the model.

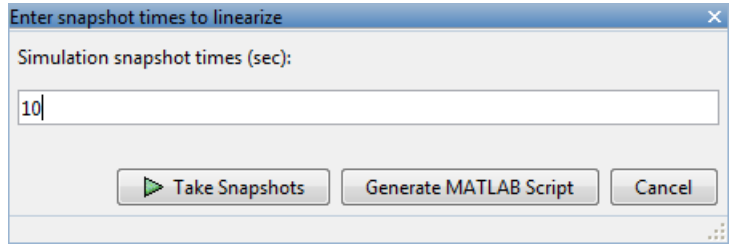
In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

3 In the Simulink model window, define the portion of the model to linearize:

- Right-click the `PID Controller` block output signal (input signal to the plant model). Select **Linear Analysis Points > Input Perturbation**.
- Right-click the `Water-Tank System` output signal, and select **Linear Analysis Points > Open-loop Output**.

4 Create a new simulation-snapshot operating point at which to linearize the model. In the Linear Analysis Tool, in the **Operating Point** drop-down list, select **Take simulation snapshot**.

5 In the Enter snapshot times to linearize dialog box, in the **Simulation Snapshot Times** field, enter one or more snapshot times at which to linearize. For this example, enter `10` to extract the operating point at this simulation time.



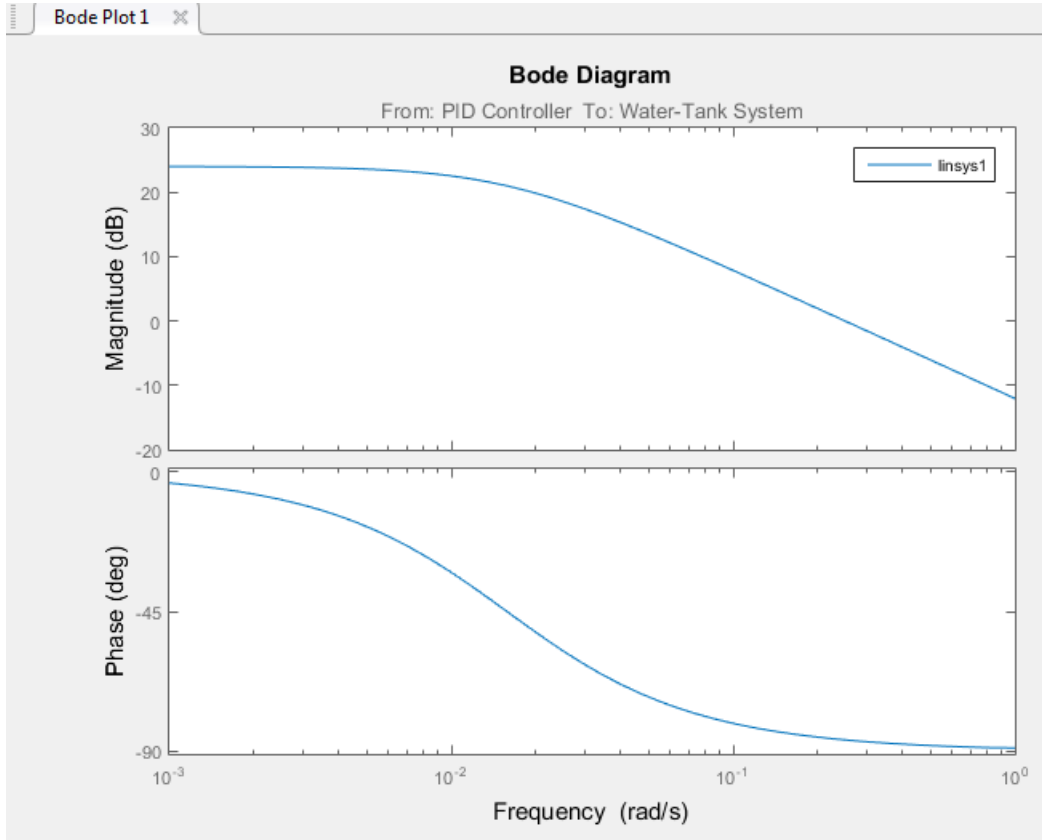
Tip To linearize the model at several operating points, specify a vector of simulation times in the **Simulation Snapshot Times** field. For example, entering `[1 10]` results in an array of two linear models, one linearized at `t = 1` and the other at `t = 10`.

6 Generate the simulation-snapshot operating point. Click **Take Snapshots**.

The operating point `op_snapshot1` appears in the Linear Analysis Workspace. In the **Operating Point** drop-down list, this operating point is now selected as the operating point to be used for linearization.

7 Linearize the model at the specified operating point and generate a bode plot of the result.

Click **Bode**. The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the Linear Analysis Workspace.



8 Double click `linsys1` in the **Linear Analysis Workspace** to see the state space representation of the linear model. Right-click on the plot and select information from the **Characteristics** menu to examine characteristics of the linearized response.

9 Close Simulink model.

```
bdclose(sys);
```

See Also

More About

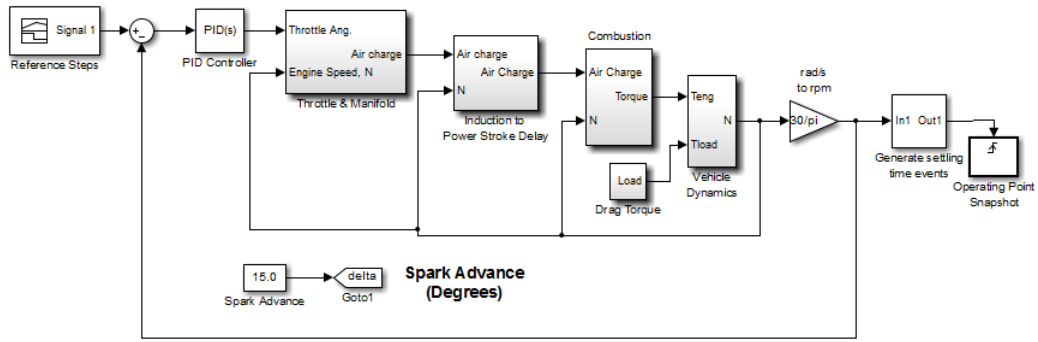
- “Linearize at Triggered Simulation Events” on page 2-95
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-110
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-117
- “Plotting Linear System Characteristics of a Chemical Reactor” on page 2-121

Linearize at Triggered Simulation Events

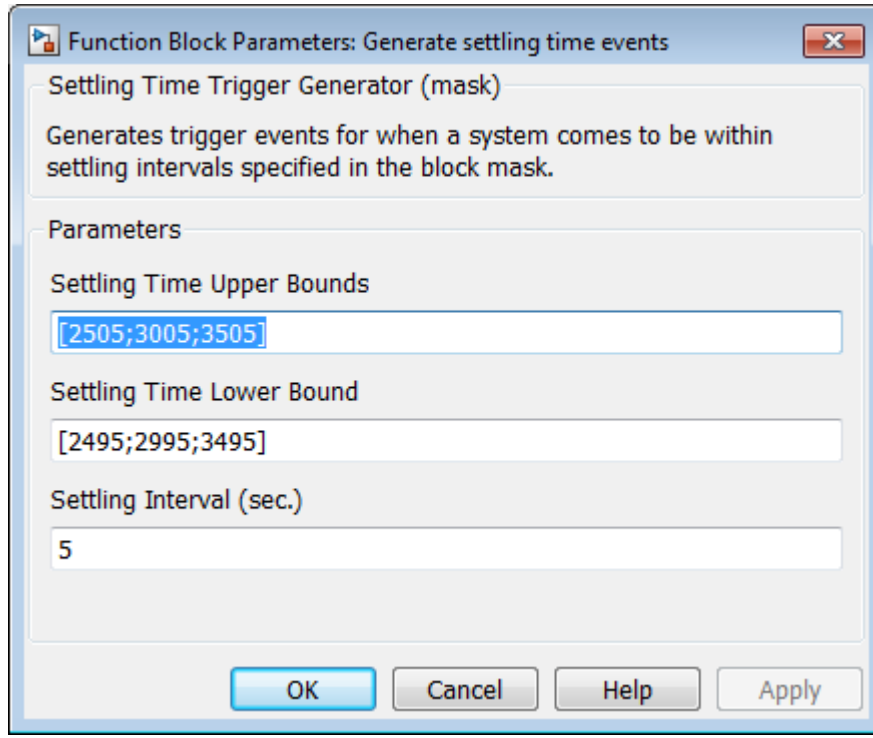
This example shows how to use the Linear Analysis Tool to linearize a model at specific events in time. Linearization events can be trigger-based events or function-call events. Specifically, the model is linearized at the steady-state operating points 2500, 3000, and 3500 rpm.

1 Open Simulink model.

```
sys = 'scdspeedtrigger';
open_system(sys)
```



To help identify when the system is at steady state, the Generate settling time events block generates settling events. This block sends rising edge trigger signals to the Operating Point Snapshot block when the engine speed settles near 2500, 3000, and 3500 rpm for a minimum of five seconds.



The model already includes a Trigger-Based Operating Point Snapshot block from the Simulink Control Design library. This block linearizes the model when it receives rising edge trigger signals from the Generate settling time events block.

2 Compute the steady-state operating point at `60` time units.

```
op = findop(sys,60);
```

This command simulates the model for `60` time units, and extracts the operating points at each simulation event that occurs during this time interval.

3 Define the portion of the model to linearize.

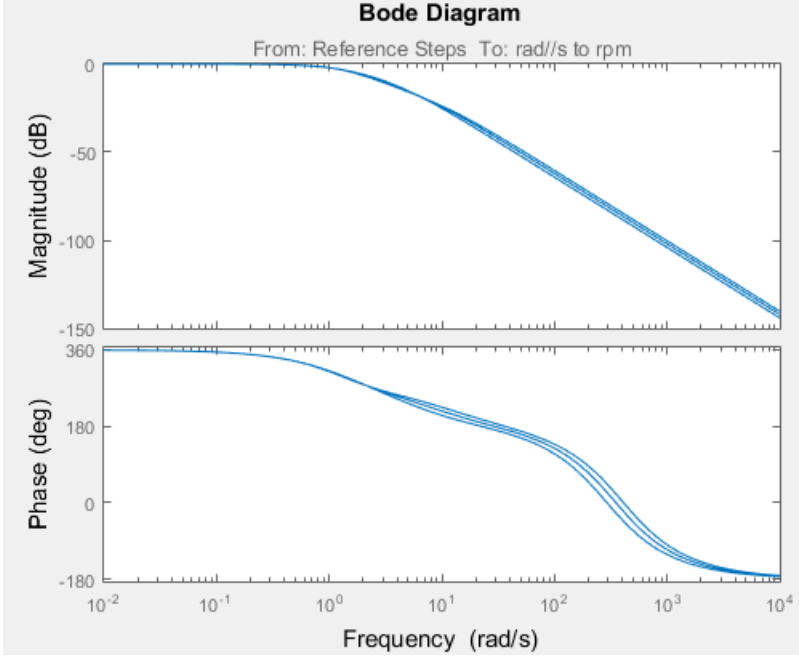
```
io(1) = linio('scdspeedtrigger/Reference Steps',1,'input');
io(2) = linio('scdspeedtrigger/rad//s to rpm',1,'output');
```

4 Linearize the model.

```
linsys = linearize(sys,op(1:3),io);
```

5 Compare linearized models at 2500, 3000, and 3500 rpm using Bode plots of the closed-loop transfer functions.

```
bode(linsys);
```



See Also

Functions

`findop`

Blocks

Trigger-Based Operating Point Snapshot

More About

- “Linearize at Simulation Snapshot” on page 2-91
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-110

Linearization of Models with Delays

This example shows how to linearize a Simulink model with delays in it.

Linearization of Models with Continuous Delays

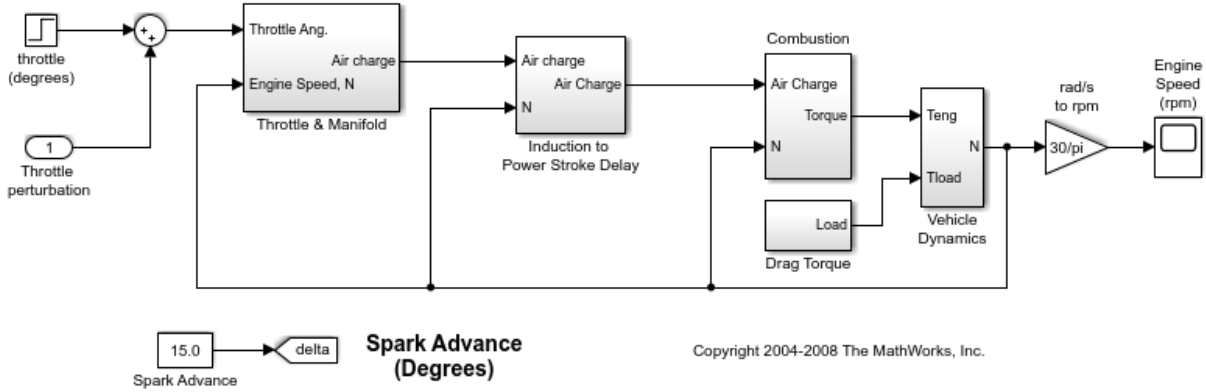
You can linearize a Simulink model with continuous time delays blocks such as the Transport Delay, Variable Transport Delay, and Variable Time Delay using one of the following options:

- Use a Pade approximations of the delays to get a rational linear system through linearizations.
- Compute a linearization where the delay is exactly represented. Use this option when you need accurate simulation and frequency responses from a linearized model and when assessing the accuracy of Pade approximation.

By default, Simulink Control Design uses Pade approximations of the delay blocks in a Simulink model.

To open the engine speed model used in this example, type

```
model = 'scdspeed';
open_system(model);
```



The engine speed model contains a Variable Transport Delay block named dM/dt in the subsystem Induction to Power Stroke Delay. For convenience you can store the path to the block in a MATLAB variable by typing

```
DelayBlock = 'scdspeed/Induction to  Power Stroke Delay/dM//dt delay';
```

To compute a linearization using a first order approximation, use one of the following techniques to set the order of the Pade approximation to 1:

- In the Variable Transport Delay block dialog box, enter 1 in the **Pade Order (for linearization)** field.
- At the command line, enter the following command:

```
set_param(DelayBlock,'PadeOrder','1');
```

Next, specify the linearization I/O to throttle angle as the input and engine speed as the output by running:

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
```

Compute the linearization using the following linearize command:

```
sys_1st_order_approx = linearize(model,io);
```

You can compute a linearization using a second order approximation by setting the Pade order to 2:

```
set_param(DelayBlock,'PadeOrder','2');
sys_2nd_order_approx = linearize(model,io);
```

To compute a linear model with the exact delay representation, set the 'UseExactDelayModel' property in the linoptions object to on:

```
opt = linearizeOptions;
opt.UseExactDelayModel = 'on';
```

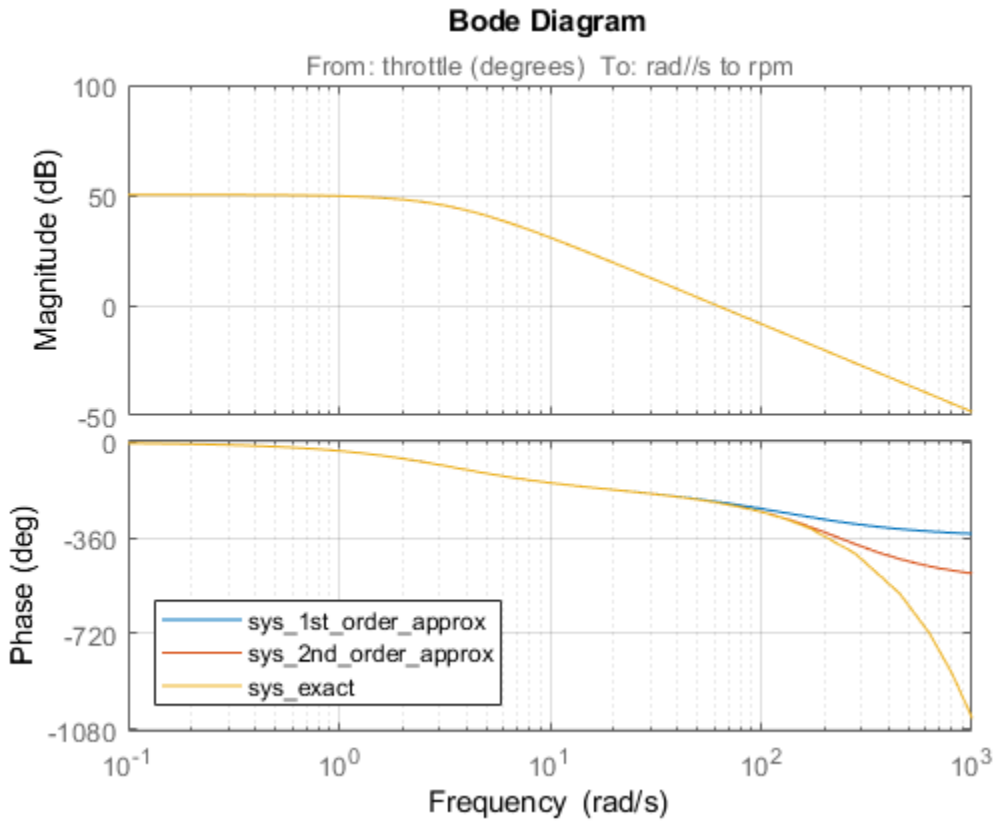
Linearize the model using the following linearize command:

```
sys_exact = linearize(model,io,opt);
```

Compare the Bode response of the Pade approximation model and the exact linearization model by running:

```
p = bodeoptions('cstprefs');
p.Grid = 'on';
p.PhaseMatching = 'on';
p.XLimMode = {'Manual'};
```

```
p.XLim = {[0.1 1000]};
f = figure;
bode(sys_1st_order_approx,sys_2nd_order_approx,sys_exact,p);
h = legend('sys_1st_order_approx','sys_2nd_order_approx','sys_exact',...
           'Location','SouthWest');
h.Interpreter = 'none';
```



In the case of a first order approximation, the phase begins to diverge around 50 rad/s and diverges around 100 rad/s.

Close the Simulink model.

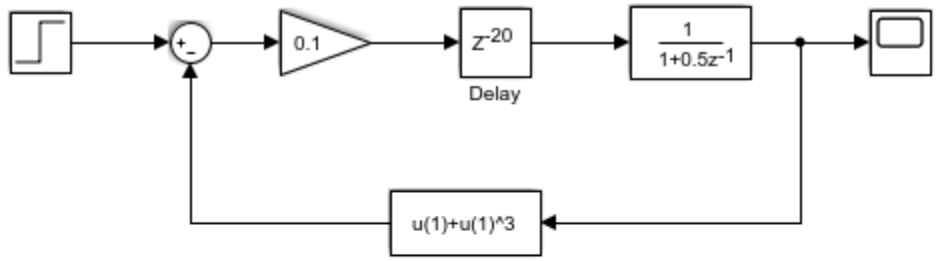
```
bdclose(model)
```

Linearization of Models with Discrete Delays

When linearizing a model with discrete delay blocks, such as (Integer) Delay and Unit Delay blocks use the exact delay option to account for the delays without adding states to the model dynamics. Explicitly accounting for these delays improves your simulation performance for systems with many discrete delays because your fewer states in your model.

To open the Simulink model of a discrete system with a Delay block with 20 delay state used for this example, run the following.

```
model = 'scdintegerdelay';
open_system(model);
```



By default the linearization includes all of the states folded into the linear model. Set the linearization I/Os and linearize the model as follows:

```
io(1) = linio('scdintegerdelay/Step',1,'input');
io(2) = linio('scdintegerdelay/Discrete Filter',1,'output');
sys_default = linearize(model,io);
```

Integrate the resulting model to see that it has 21 states (1 - Discrete Filter, 20 - Integer Delay).

```
size(sys_default)
```

```
State-space model with 1 outputs, 1 inputs, and 21 states.
```

You can linearize this same model using the 'UseExactDelayModel' property as follows:

```
opt = linearizeOptions;
opt.UseExactDelayModel = 'on';
sys_exact = linearize(model,io,opt);
```

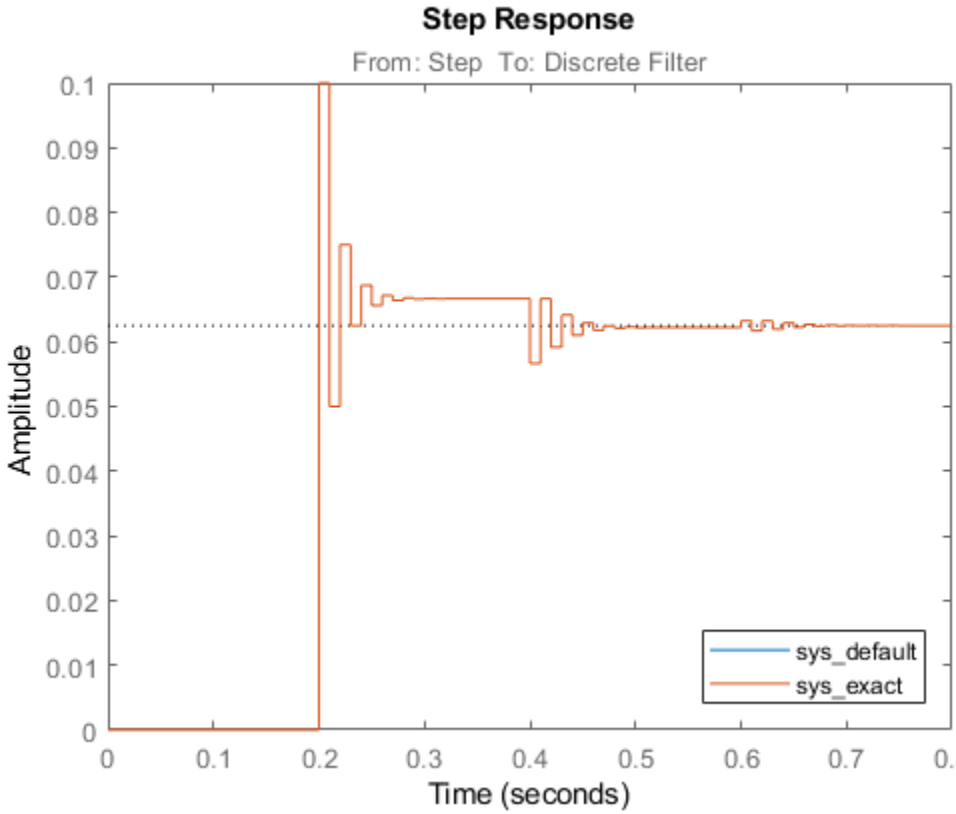
Interrogating the new resulting model shows that it has 1 state and the delays are accounted for internally in the linearized model.

```
size(sys_exact)
```

```
State-space model with 1 outputs, 1 inputs, and 1 states.
```

Run a step response simulation of both linearized model to see that they are identical by running the following commands.

```
step(sys_default,sys_exact);
h = legend('sys_default','sys_exact',...
           'Location','SouthEast');
h.Interpreter = 'none';
```



Close the Simulink model and clean up figures.

```
bdclose(model)
close(f)
```

Working with Linearized Models with Delays

For more information on manipulating linearized models with delays, see the Control System Toolbox documentation along with the examples "Specifying Time Delays" and "Analyzing Control Systems with Delays" .

See Also

`linearize` | `linearizeOptions`

More About

- “Models with Time Delays” on page 2-168

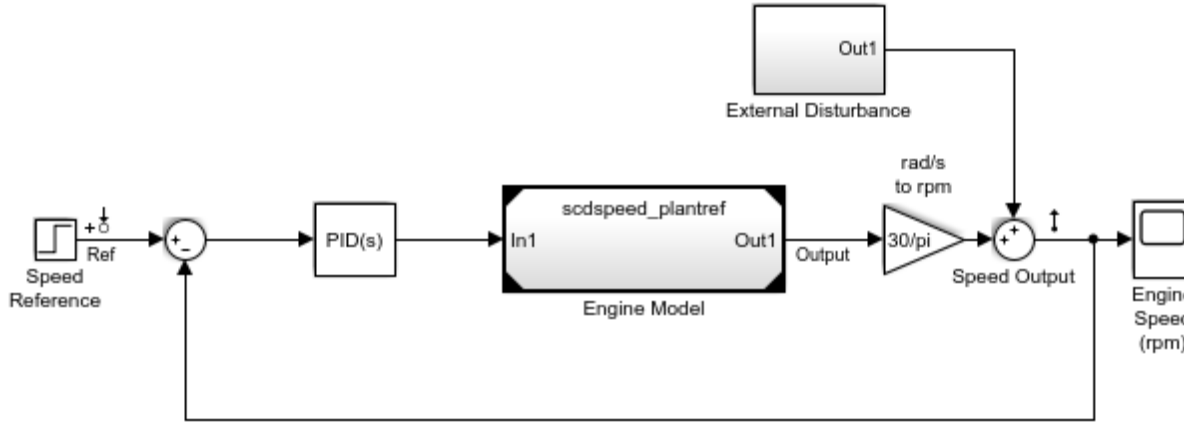
Linearization of Models with Model References

This example shows the features available in Simulink Control Design for linearizing models containing references to other models with a Model block.

Linear Analysis

The model `scdspeed_ctrlloop` is a componentized version of the model `scdspeedctrl`. In this model, the engine speed model is a component represented in the model `scdspeed_plantref` which is referenced using a model reference block. To open the engine model, run:

```
topmdl = 'scdspeed_ctrlloop';
open_system(topmdl);
```



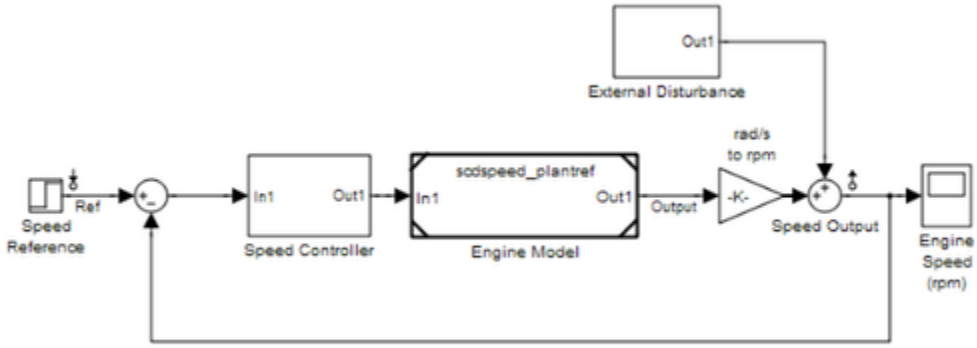
Initially, the reference is set to run its simulation in accelerator mode. The accelerator simulation mode is indicated by the black triangles on the corners of the model block `scdspeed_ctrlloop/Engine Model`.

By default, Engine Model reference block is set to accelerator simulation mode, as indicated by the block triangles on the block. Linearizing the model with this block set to accelerator simulation mode numerically perturbs the entire Engine Model block. The accuracy of this linearization is very sensitive to the blocks within the Engine model. In particular, the variable transport delay block is very problematic.

To achieve an accurate linearization, set the model reference block to normal simulation mode to allow the block-by-block linearization of the referenced model by running the following command.

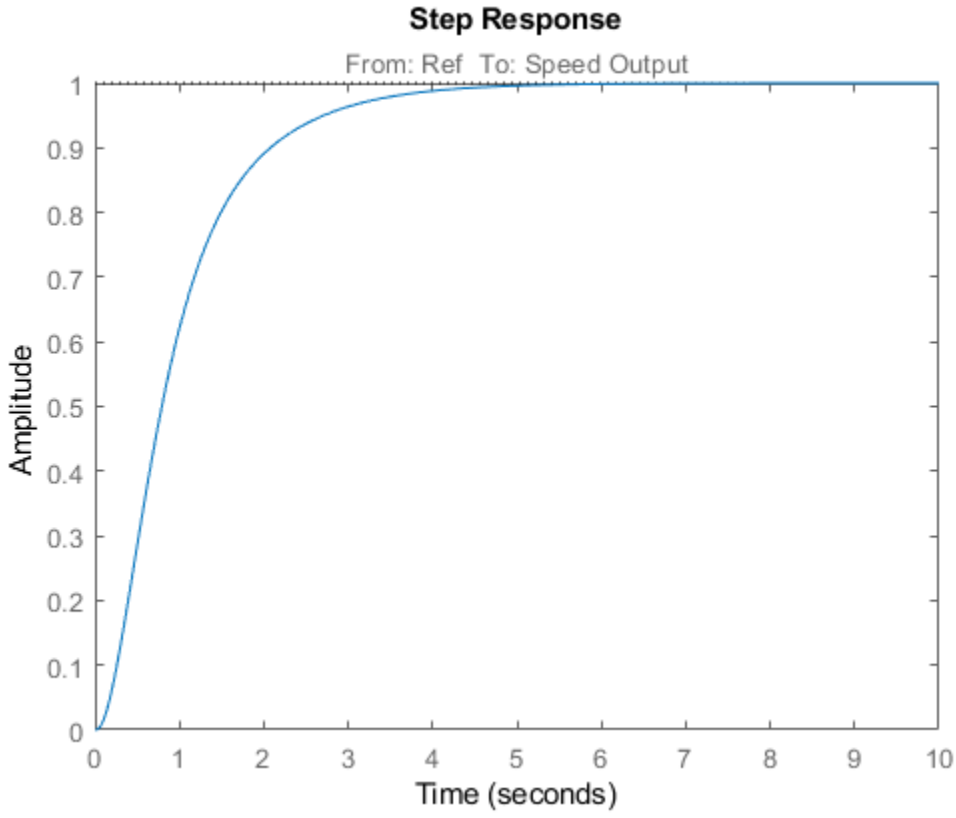
```
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')
```

Notice that the corners of the model block are now white triangles indicating that its simulation mode is set to normal, as showing in the following figure.



Linearize the model between the speed reference and the speed output and plot the resulting step response by running:

```
io(1) = linio('scdspeed_ctrlloop/Speed Reference',1,'input');
io(2) = linio('scdspeed_ctrlloop/Speed Output',1,'output');
sys_normal = linearize(topmdl,io);
step(sys_normal);
```



Close the Simulink model.

```
bdclose('scdspeed_ctrlloop');
```

Another benefit of switching the model reference to normal mode simulation is that you can take advantage of the exact delay representations.

For more information on linearizing models with delays see the example "Linearizing Models with Delays".

See Also

`linearize`

Visualize Linear System at Multiple Simulation Snapshots

This example shows how to visualize linear system characteristics of a nonlinear Simulink model at multiple simulation snapshots.

1 Open Simulink model.

For example:

```
watertank
```

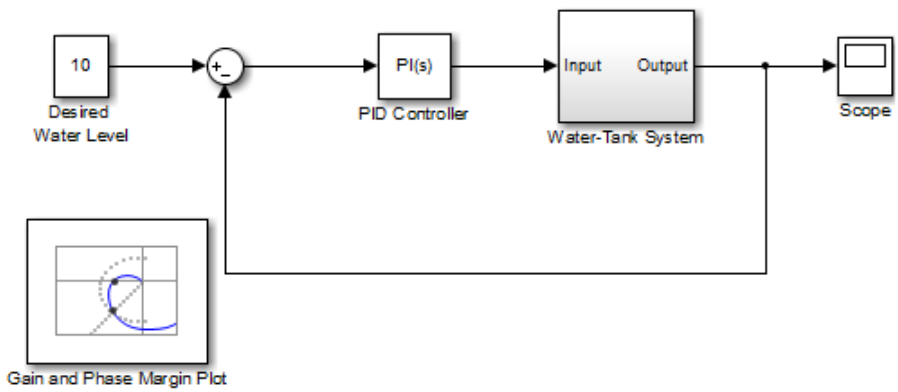
2 Open the Simulink Library Browser by selecting **View > Library Browser** in the model window.

3 Add a plot block to the Simulink model.

a In the **Simulink Control Design** library, select **Linear Analysis Plots**.

b Drag and drop a block, such as the Gain and Phase Margin Plot block, into the Simulink model window.

The model now resembles the following figure.



4 Double-click the block to open the Block Parameters dialog box.

To learn more about the block parameters, see the block reference pages.

5 Specify the linearization I/O points.

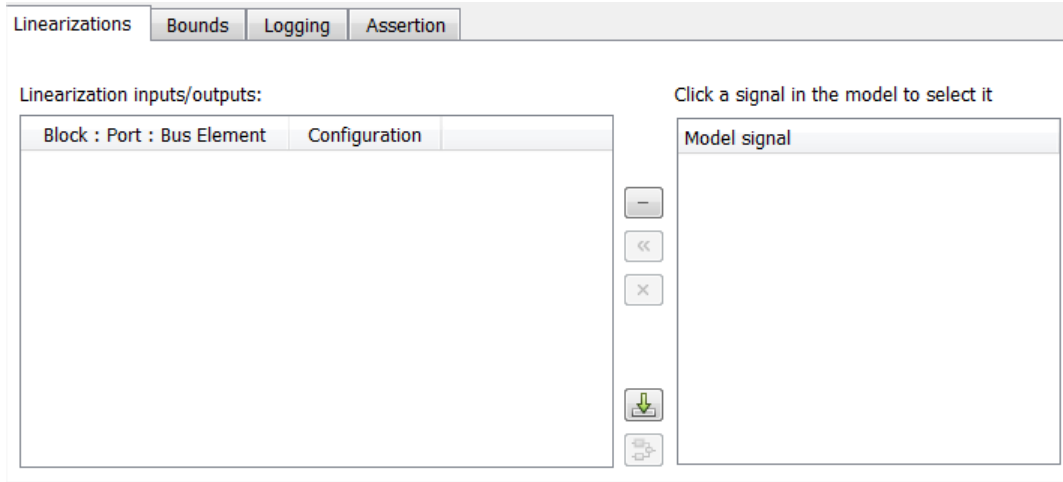
The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

a To specify an input:

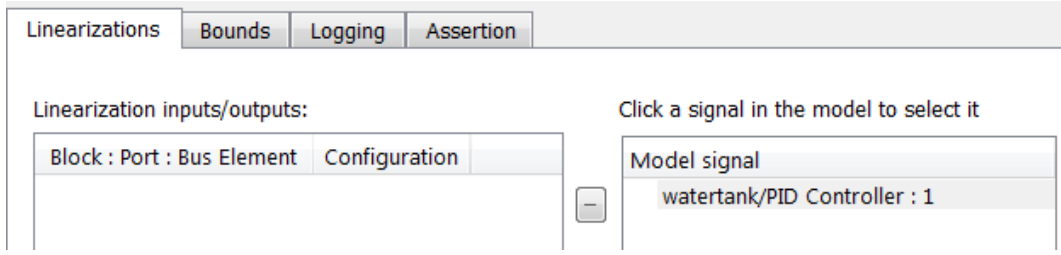
i Click  adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.



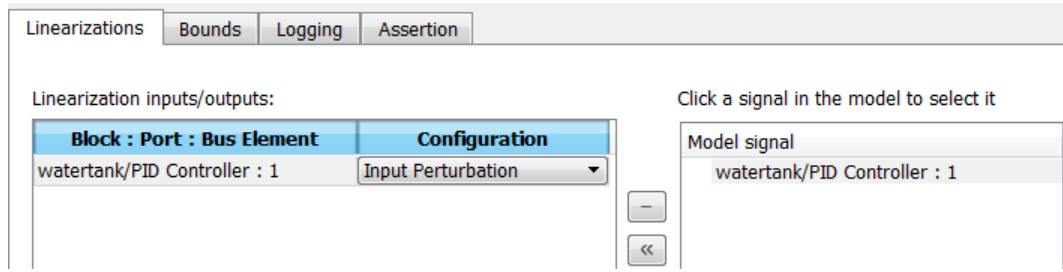
ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

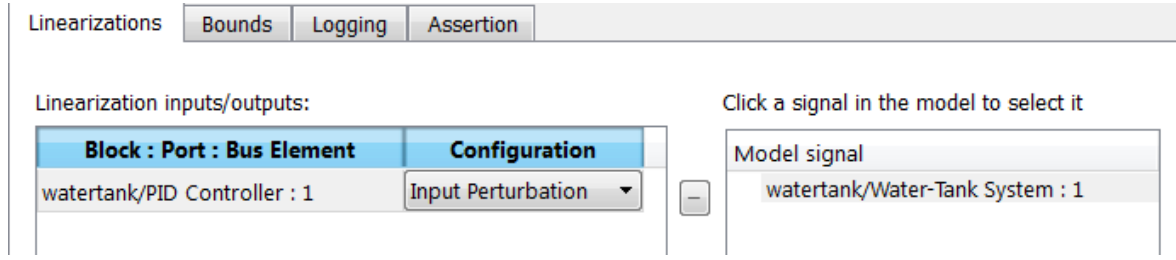
iii Click « to add the signal to the **Linearization inputs/outputs** table.



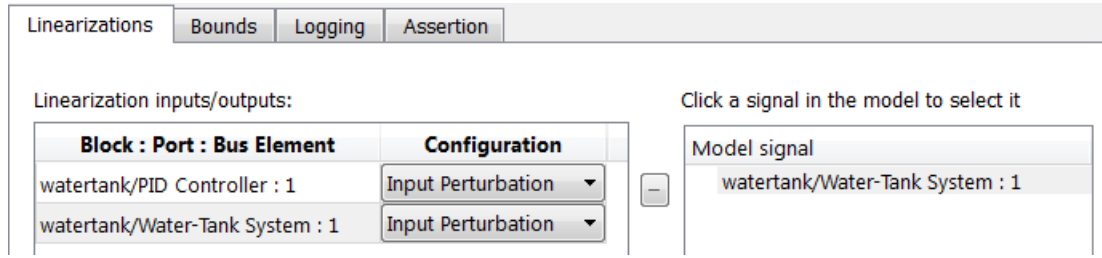
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.

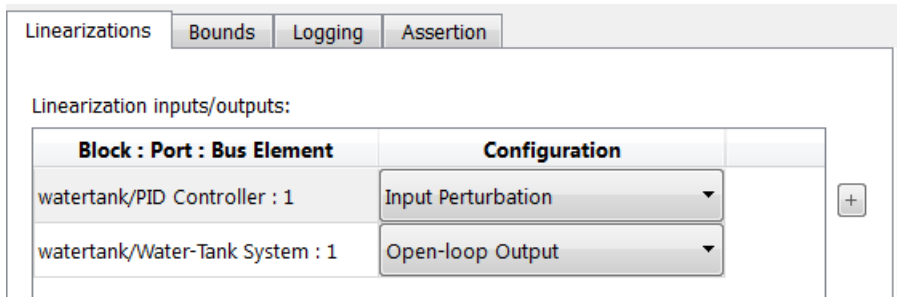


ii Click « to add the signal to the **Linearization inputs/outputs** table.



iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



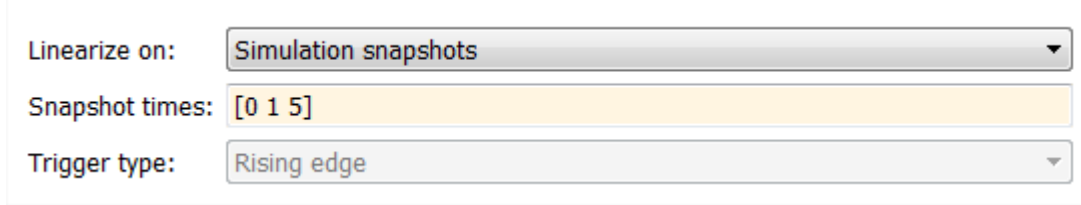
c Click  to collapse the **Click a signal in the model to select it** area.

Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear in the **Linearization inputs/outputs** table.

6 Specify simulation snapshot times.

a In the **Linearizations** tab, verify that `Simulation snapshots` is selected in **Linearize on**.

b In the **Snapshot times** field, type `[0 1 5]`.



7 Specify a plot type to plot the gain and phase margins. The plot type is `Bode` by default.

a Select `Nichols` in **Plot type**

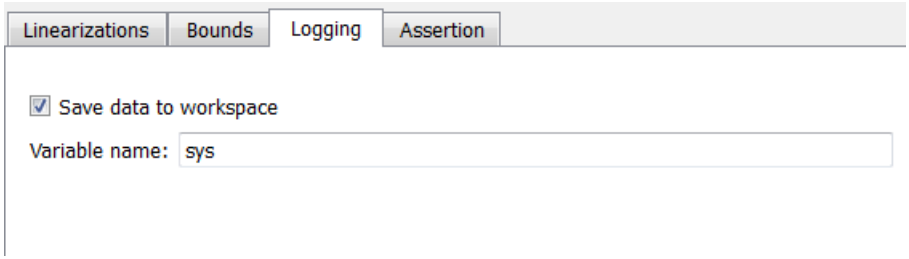
b Click **Show Plot** to open an empty Nichols plot.

8 Save the linear system.

a Select the **Logging** tab.

b Select the **Save data to workspace** option and specify a variable name in the **Variable name** field.

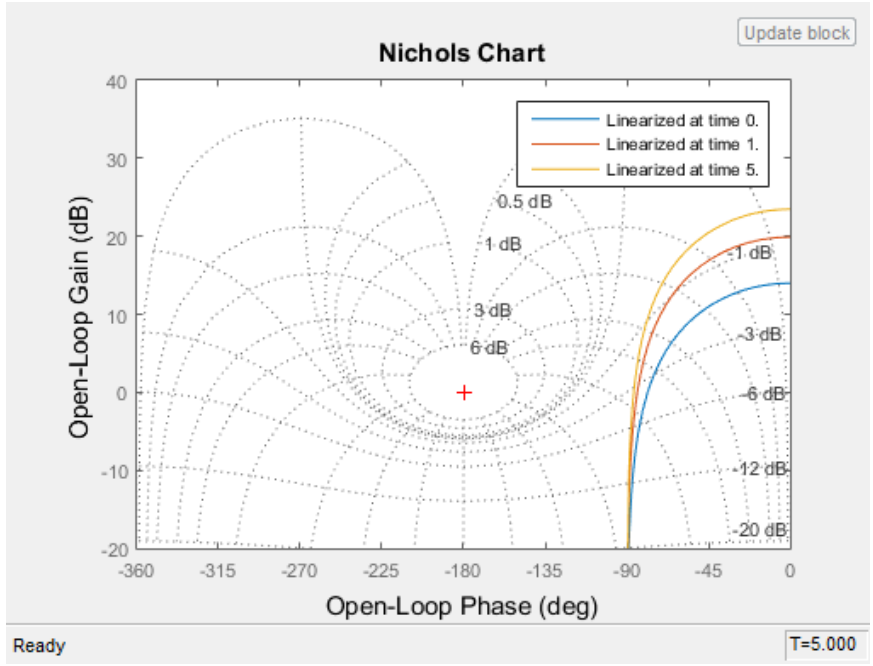
The **Logging** tab now resembles the following figure.



9 Plot the gain and phase margins by clicking  in the plot window.

The software linearizes the portion of the model between the linearization input and output at the simulation times of 0, 1 and 5 and plots gain and phase margins.

After the simulation completes, the plot window resembles the following figure.



Tip Click  to view the legend.

The computed linear system is saved as `sys` in the MATLAB workspace. `sys` is a structure with `time` and `values` fields. To view the structure, type:

```
sys
```

This command returns the following results:

```
sys =

         time: [3x1 double]
       values: [4-D ss]
    blockName: 'watertank/Gain and Phase Margin Plot'
```

- The `time` field contains the simulation times at which the model is linearized.
- The `values` field is an array of state-space objects (Control System Toolbox) which store the linear systems computed at the specified simulation times.

(If the Simulink model is configured to save simulation output as a single object, the data structure `sys` is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.)

See Also

Gain and Phase Margin Plot

More About

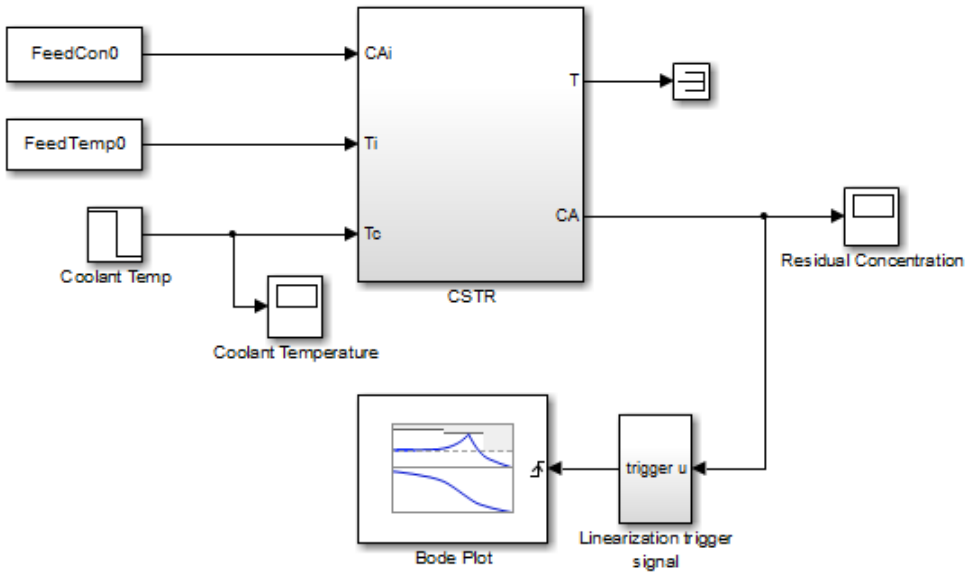
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77
- "Plotting Linear System Characteristics of a Chemical Reactor" on page 2-121
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-117
- "Linearize at Simulation Snapshot" on page 2-91
- "Linearize at Triggered Simulation Events" on page 2-95

Visualize Linear System of a Continuous-Time Model Discretized During Simulation

This example shows how to discretize a continuous-time model during simulation and plot the model's discretized linear behavior.

1 Open the Simulink model:

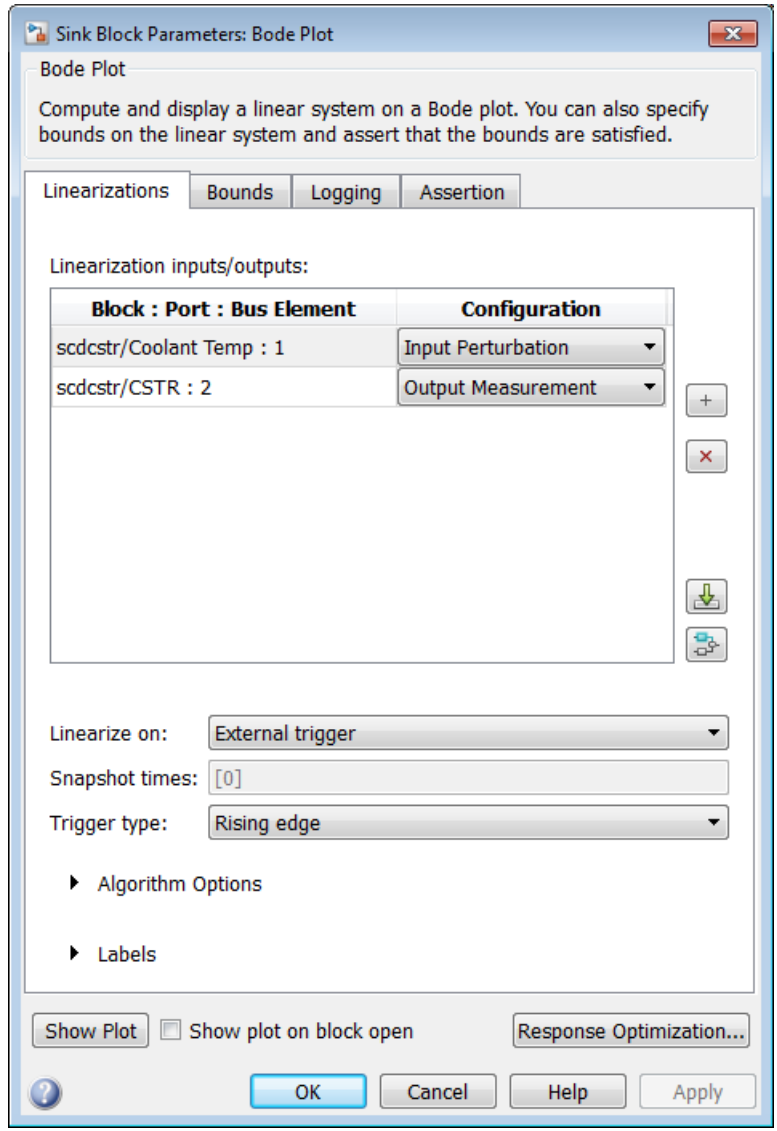
```
scdcstr
```



In this model, the Bode Plot block has already been configured with:

- Input point at the coolant temperature input `Coolant Temp`
- Output point at the residual concentration output `CA`
- Settings to linearize the model on a rising edge of an external trigger. The trigger signal is modeled in the `Linearization trigger signal` block in the model.
- Saving the computed linear system in the MATLAB workspace as `LinearReactor`.

To view these configurations, double-click the block.

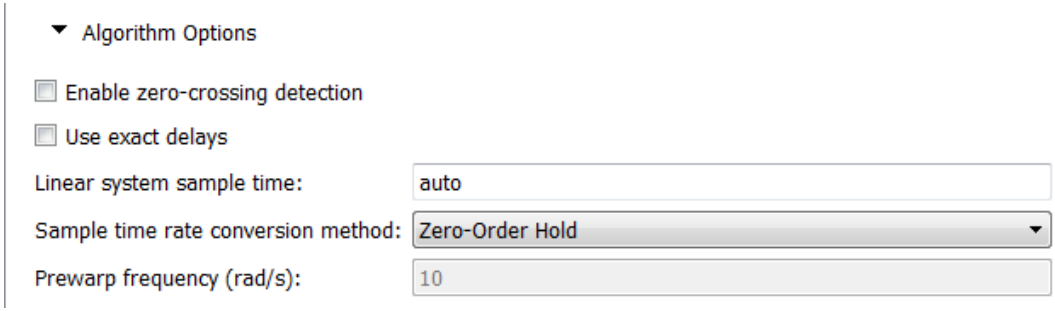


To learn more about the block parameters, see the block reference pages.

2 Specify the sample time to compute the discrete-time linear system.

a Click ▶ adjacent to **Algorithm Options**.

The option expands to display the linearization algorithm options.



b Specify a sample time of 2 in the **Linear system sample time** field.

To learn more about this option, see the block reference page.

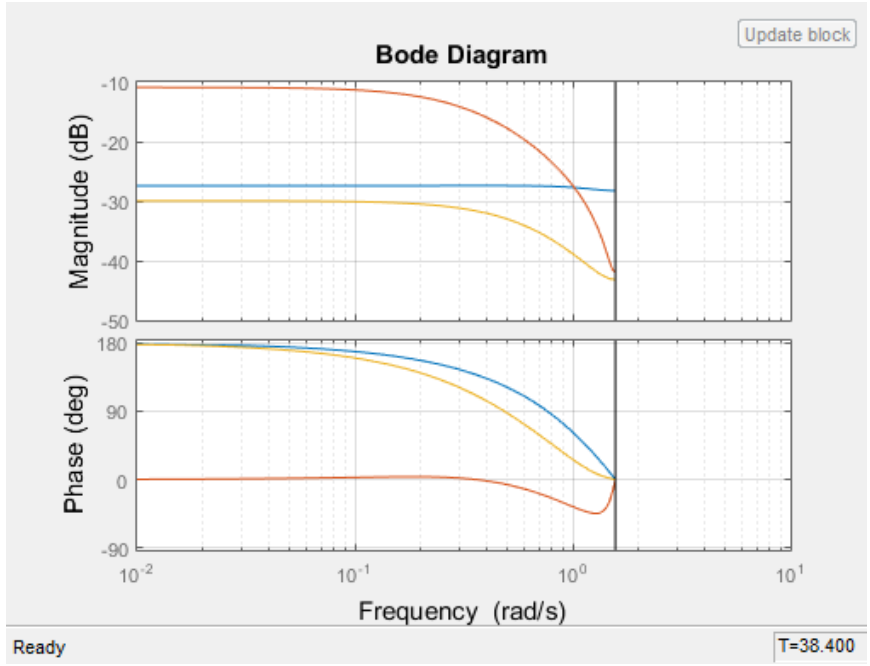
3 Click **Show Plot** to open an empty Bode plot window.

4 Plot the Bode magnitude and phase by clicking ▶ in the plot window.

During simulation, the software:

- Linearizes the model on encountering a rising edge.
- Converts the continuous-time model into a discrete-time linear model with a sample time of 2. This conversion uses the default `Zero-Order Hold` method to perform the sample time conversion.

  The software plots the discrete-time linear behavior in the Bode plot window. After the simulation completes, the plot window resembles the following figure.



The plot shows the Bode magnitude and phase up to the Nyquist frequency, which is computed using the specified sample time. The vertical line on the plot represents the Nyquist frequency.

See Also

More About

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-110
- "Plotting Linear System Characteristics of a Chemical Reactor" on page 2-121
- "Linearize at Simulation Snapshot" on page 2-91
- "Linearize at Triggered Simulation Events" on page 2-95

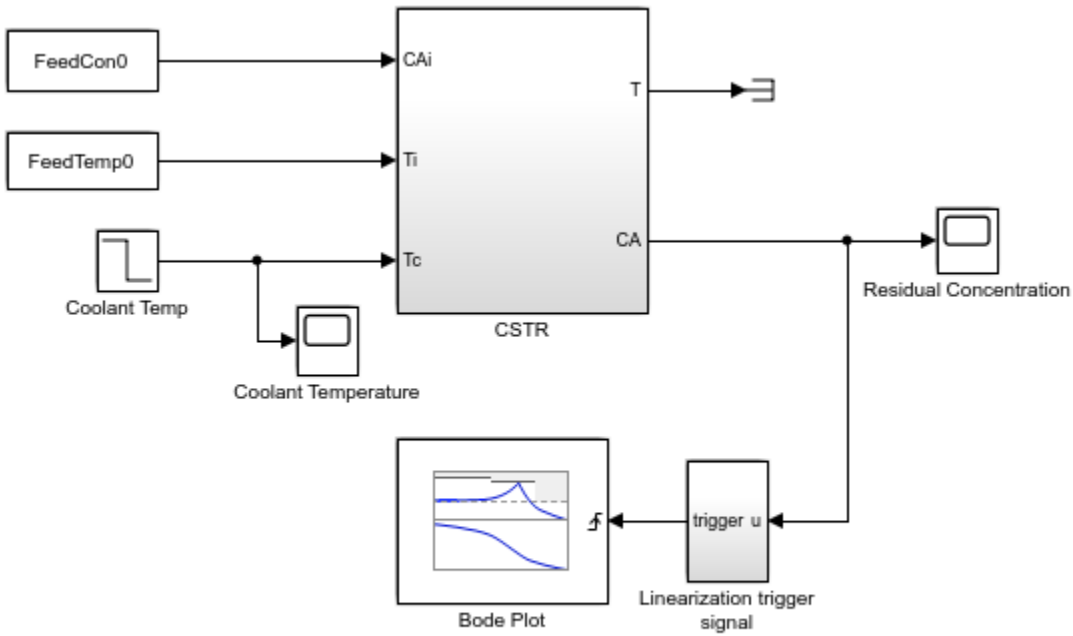
Plotting Linear System Characteristics of a Chemical Reactor

This example shows how to plot linearization of a Simulink model at particular conditions during simulation. The Simulink Control Design software provides blocks that you can add to Simulink models to compute and plot linear systems during simulation. In this example, a linear system of a continuous-stirred chemical reactor is computed and plotted on a Bode plot as the reactor transitions through different operating points.

Chemical Reactor Model

Open the Simulink model of the chemical reactor:

```
open_system('scdcstr')
```



The reactor has three inputs and two outputs:

- The `FeedCon0`, `FeedTemp0` and `Coolant Temp` blocks model the feed concentration, feed temperature, and coolant temperature inputs respectively.
- The `T` and `CA` ports of the `CSTR` block model the reactor temperature and residual concentration outputs respectively.

This example focuses on the response from coolant temperature, `Coolant Temp`, to residual concentration, `CA`, when the feed concentration and feed temperature are constant.

For more information on modeling reactors, see Seborg, D.E. et al., "Process Dynamics and Control", 2nd Ed., Wiley, pp.34-36.

Plotting the Reactor Linear Response

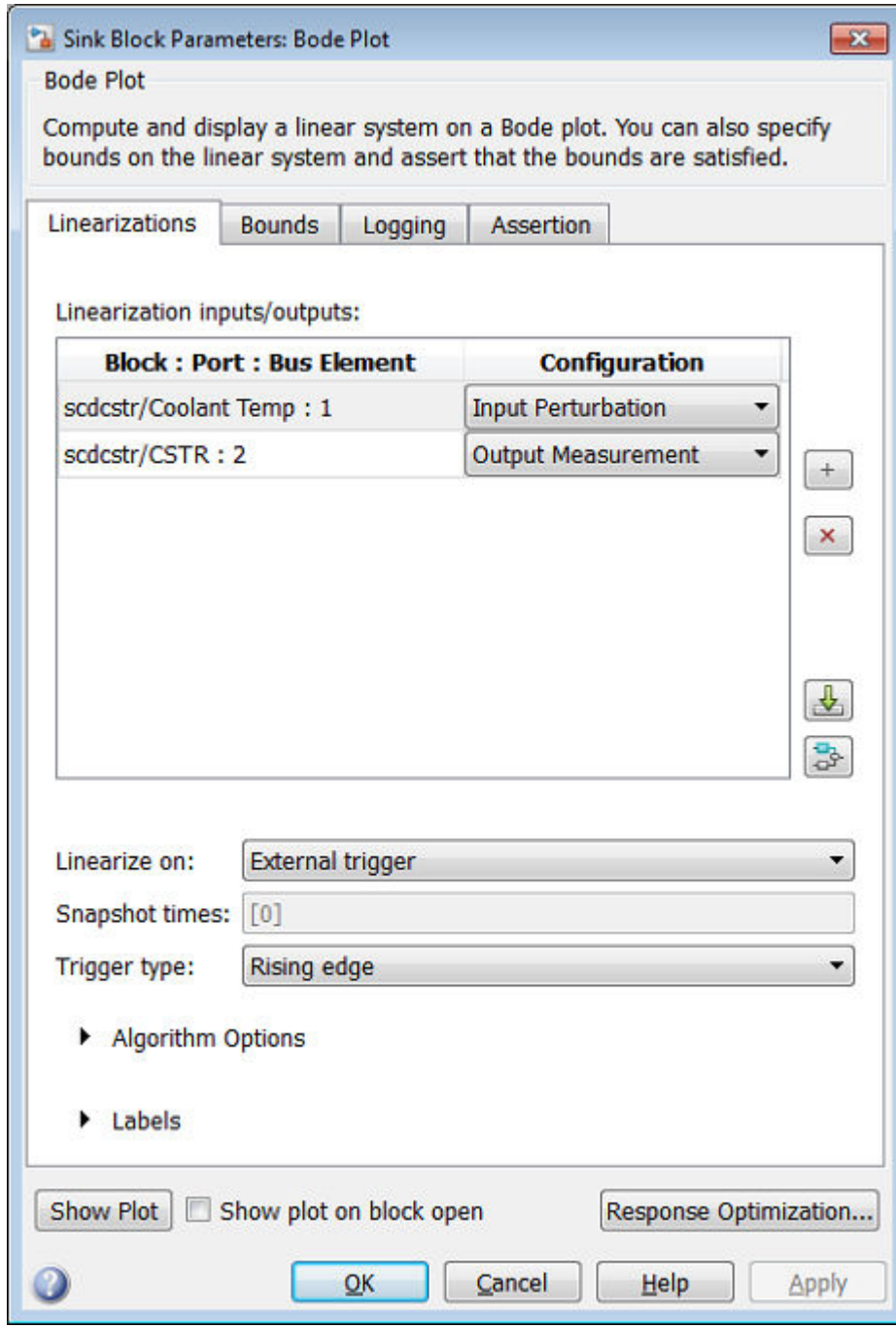
The reactor model contains a `Bode Plot` block from the Simulink Control Design Linear Analysis Plots library. The block is configured with:

- A linearization input at the coolant temperature `Coolant Temp`.
- A linearization output at the residual concentration `CA`.

The block is also configured to perform linearizations on the rising edges of an external trigger signal. The trigger signal is computed in the `Linearization trigger signal` block which produces a rising edge when the residual concentration is:

- At a steady state value of 2
- In a narrow range around 5
- At a steady state value of 9

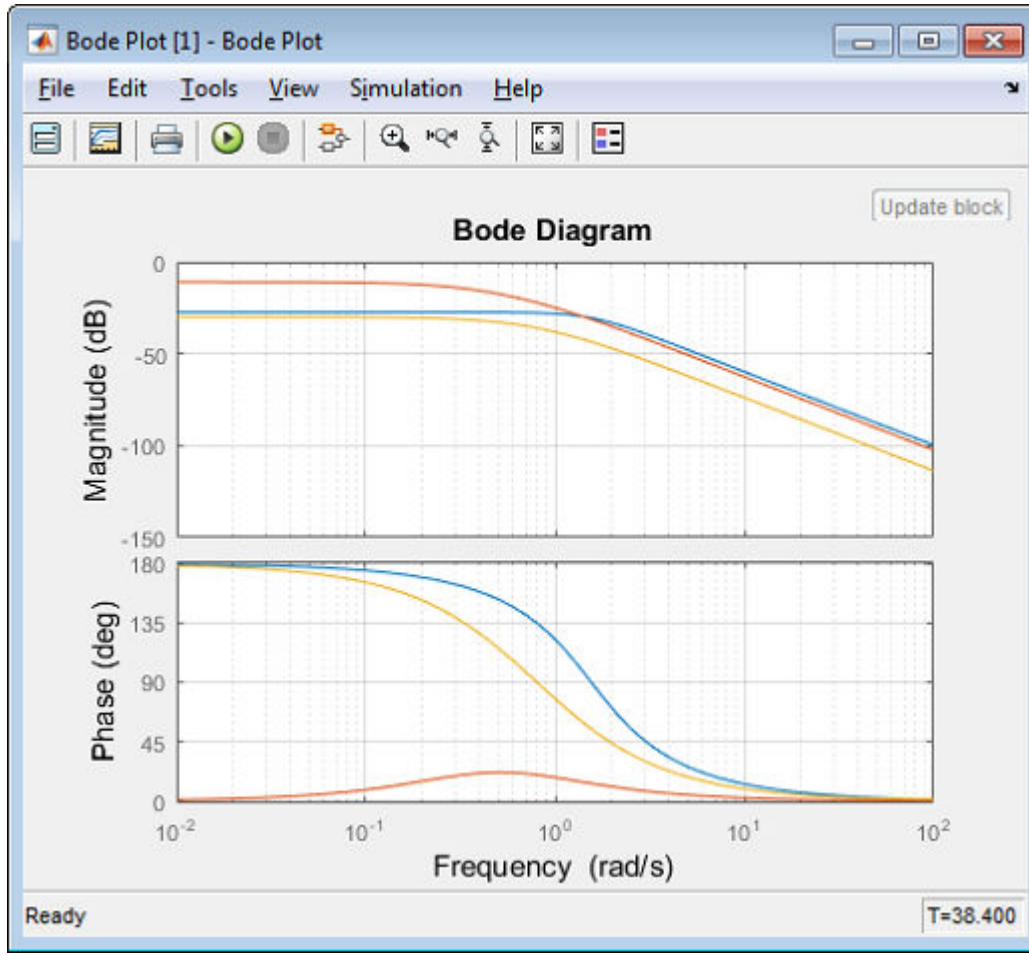
Double-clicking the `Bode Plot` block lets you view the block configuration.


Sink Block Parameters: Bode Plot
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
scdcstr/Coolant Temp : 1
Input Perturbation
scdcstr/CSTR : 2
Output Measurement
Linearize on:
External trigger
Snapshot times:
[0]
Trigger type:
Rising edge
Algorithm Options
Labels
Show Plot
Show plot on block open
Response Optimization...
OK
Cancel
Help
Apply

Clicking **Show Plot** in the Block Parameters dialog box opens a Bode Plot window which shows the response of the computed linear system from `Coolant Temp` to `CA`. To compute the linear system and view its response, simulate the model using one of the following:

- Click the `Run` button in the Bode Plot window.
- Select **Simulation > Run** in the Simulink model window.
- Type the following command:

```
sim('scdcstr')
```



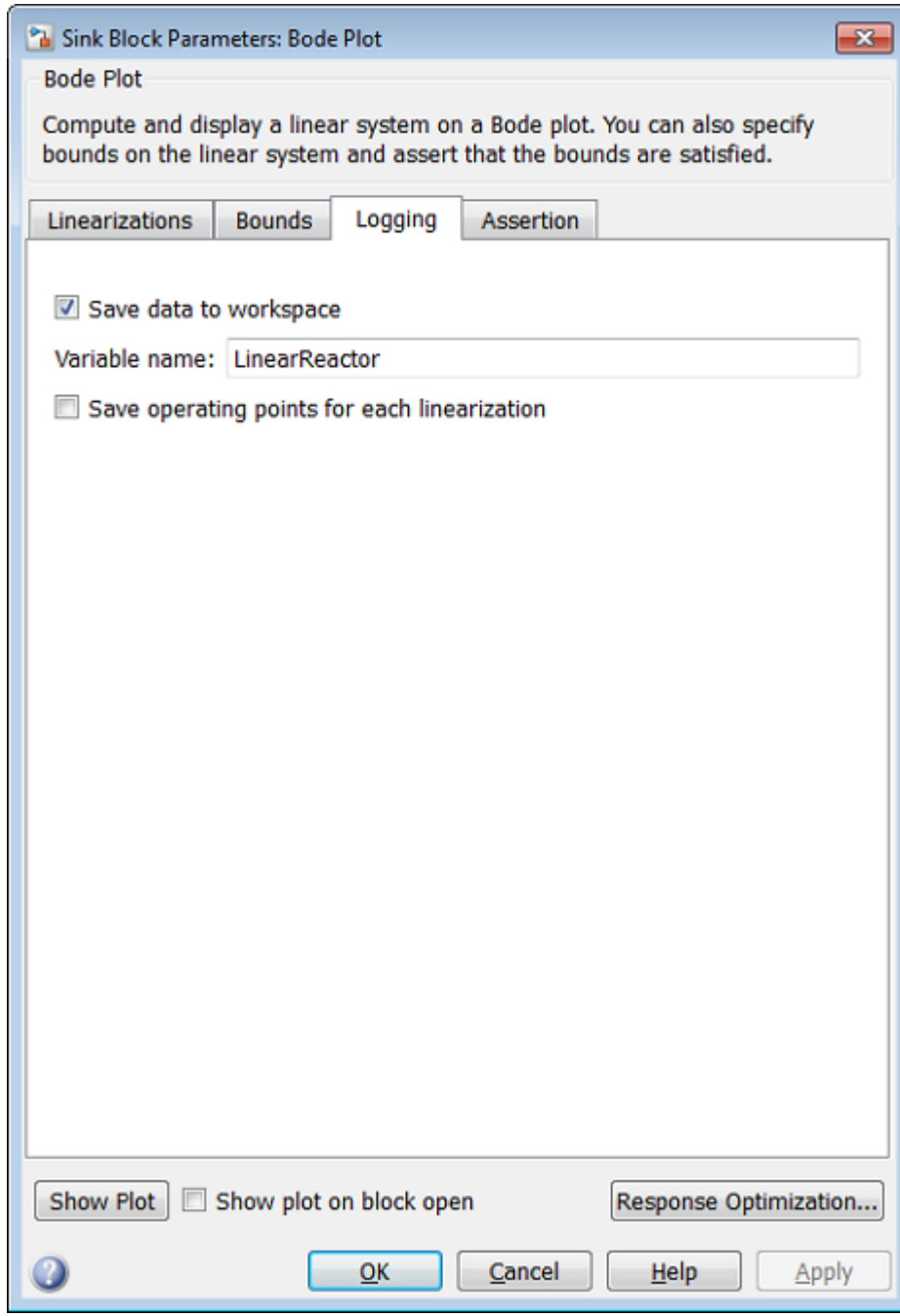
The Bode plot shows the linearized reactor at three operating points corresponding to the trigger signals defined in the `Linearization trigger signal` block:

- At 5 sec, the linearization is for a low residual concentration.
- At 38 sec, the linearization is for a high residual concentration.
- At 27 sec, the linearization is as the reactor transitions from a low to high residual concentration.

The linearizations at low and high residual concentrations are similar but the linearization during the transition has a significantly different DC gain and phase characteristics. At low frequencies, the phase differs by 180 degrees, indicating the presence of either an unstable pole or zero.

Logging the Reactor Linear Response

The **Logging** tab in the `Bode Plot` block specifies that the computed linear systems be saved as a workspace variable.


Sink Block Parameters: Bode Plot
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
Save data to workspace
Variable name:
LinearReactor
Save operating points for each linearization
Show Plot
Show plot on block open
Response Optimization...
OK
Cancel
Help
Apply

The linear systems are logged in a structure with `time` and `values` fields.

```
LinearReactor
```

```
LinearReactor =

  struct with fields:

         time: [3x1 double]
       values: [1x1x3x1 ss]
    blockName: 'scdcstr/Bode Plot'
```

The `values` field stores the linear systems as an array of LTI state-space systems (see Arrays of LTI Models) in Control System Toolbox documentation for more information).

You can retrieve the individual systems by indexing into the `values` field.

```
P1 = LinearReactor.values(:,:,1);
P2 = LinearReactor.values(:,:,2);
P3 = LinearReactor.values(:,:,3);
```

The Bode plot of the linear system at time 27 sec, when the reactor transitions from low to high residual concentration, indicates that the system could be unstable. Displaying the linear systems in pole-zero format confirms this:

```
zpk(P1)
zpk(P2)
zpk(P3)
```

```
ans =

  From input "Coolant Temp" to output "CSTR/2":
         -0.1028
  ----------------------
  (s^2 + 2.215s + 2.415)

Continuous-time zero/pole/gain model.


ans =

  From input "Coolant Temp" to output "CSTR/2":
```

```
        -0.07514
  ---------------------
  (s+0.7567) (s-0.3484)

Continuous-time zero/pole/gain model.


ans =

  From input "Coolant Temp" to output "CSTR/2":
        -0.020462
  ---------------------
  (s+0.8542) (s+0.7528)

Continuous-time zero/pole/gain model.
```

Close the Simulink model:

```
bdclose('scdcstr')
clear('LinearReactor','P1','P2','P3')
```

See Also

More About

- “Linearize at Triggered Simulation Events” on page 2-95

Order States in Linearized Model

In this section...

Control State Order of Linearized Model using Linear Analysis Tool

This example shows how to control the order of the states in your linearized model. This state order appears in linearization results.

1 Open and configure the model for linearization by specifying linearization I/Os and an operating point for linearization. You can perform this step as shown, for example, in “Linearize at Trimmed Operating Point” on page 2-85. To preconfigure the model at the command line, use the following commands.

```
sys = 'magball';
open_system(sys)
sys_io(1) = linio('magball/Controller',1,'input');
sys_io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
setlinio(sys,sys_io)
opspec = operspec(sys);
op = findop(sys,opspec);
```

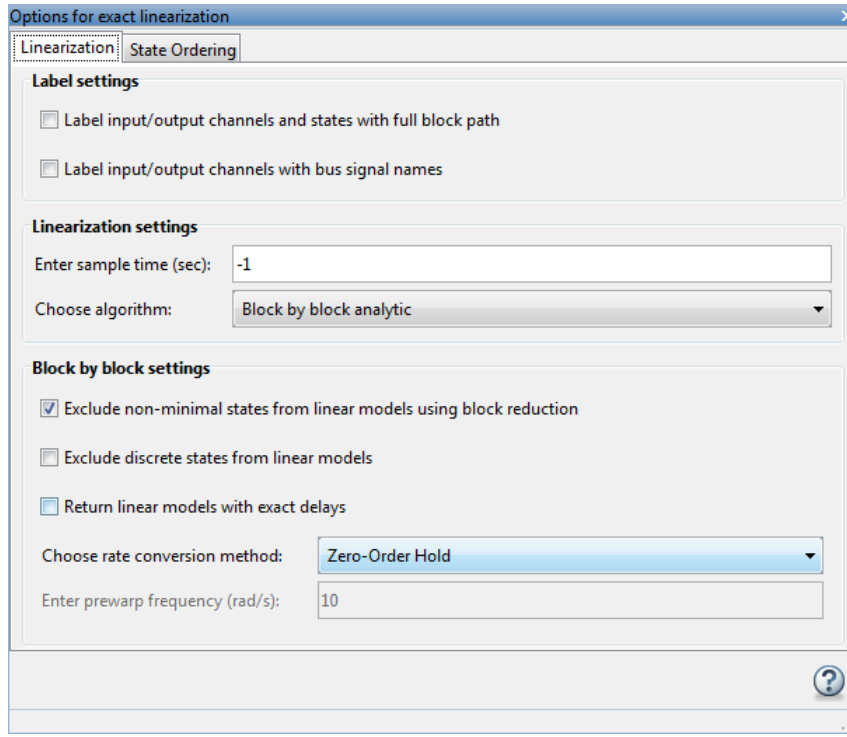
These commands specify the plant linearization and compute the steady-state operating point.

2 Open the Linear Analysis Tool for the model.

In the Simulink model window, select **Analysis** > **Control Design** > **Linear Analysis**.

3 Open the Options for exact linearization dialog box.

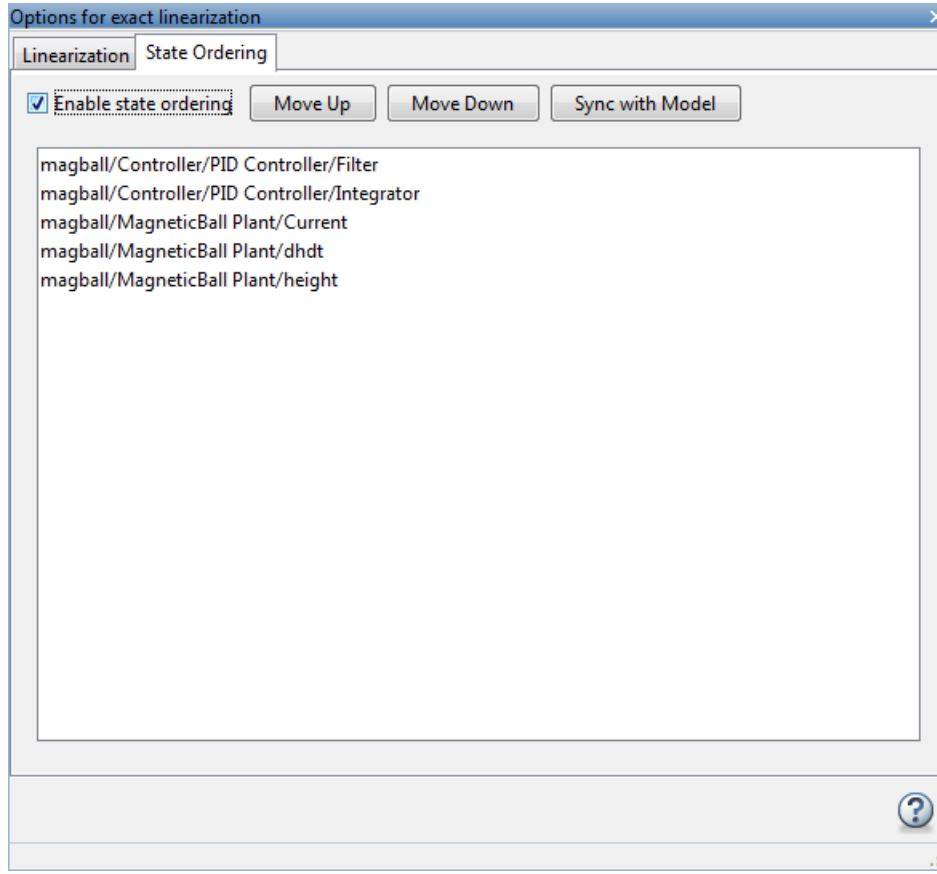
In the **Linear Analysis** tab, click  **More Options**.



4 In the **State Ordering** tab, check **Enable state ordering**.

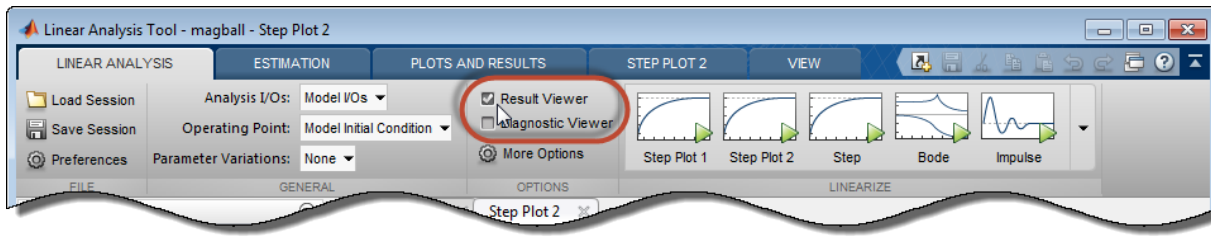
5 Specify the desired state order using the **Move Up** and **Move Down** buttons.

Tip If you change the model while its Linear Analysis Tool is open, click **Sync with Model** to update the list of states.



Click ✕ to close the dialog box.

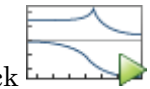
6 Enable the linearization result viewer. In the **Linear Analysis** tab, check **Result Viewer**.



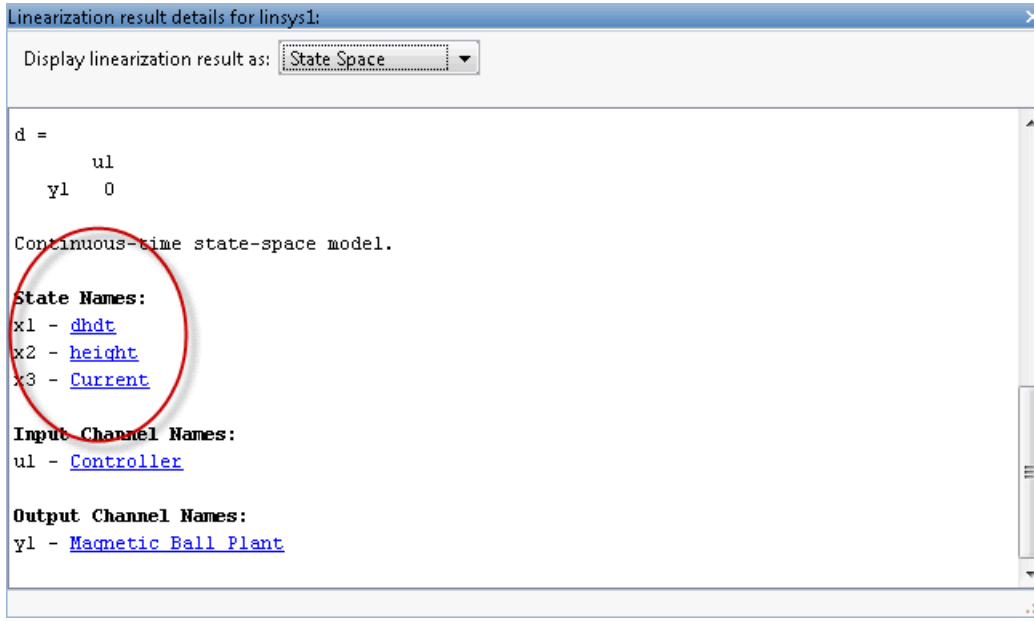
When this option is checked, the result viewer appears when you linearize the model, enabling you to view and confirm the state ordering.

Tip If you do not check **Result Viewer**, or if you close the result viewer, you can open the result viewer for a previously linearized model. To do so, in the **Plots and Results** tab, select the linear model in the Linear Analysis Workspace, and click  **Result Viewer**.

---

7 Linearize the model. For example, click  **Bode**.

A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**. The linearization result viewer opens, displaying information about that model.



The linear model states appear in the specified order.

Control State Order of Linearized Model using MATLAB Code

This example shows how to control the order of the states in your linearized model. This state order appears in linearization results.

1 Load and configure the model for linearization.

```
sys = 'magball';
load_system(sys);
sys_io(1)=linio('magball/Controller',1,'input');
sys_io(2)=linio('magball/Magnetic Ball Plant',1,'openoutput');
opspec = operspec(sys);
op = findop(sys,opspec);
```

These commands specify the plant linearization and compute the steady-state operating point.

2 Linearize the model, and show the linear model states.

```
linsys = linearize(sys,sys_io);
linsys.StateName
```

The linear model states are in default order. The linear model includes only the states in the linearized blocks, and not the states of the full model.

```
ans =
    'height'
    'Current'
    'dhdt'
```

3 Define a different state order.

```
stateorder = {'magball/Magnetic Ball Plant/height';...
              'magball/Magnetic Ball Plant/dhdt';...
              'magball/Magnetic Ball Plant/Current'};
```

4 Linearize the model again and show the linear model states.

```
linsys = linearize(sys,sys_io,'StateOrder',stateorder);
linsys.StateName
```

The linear model states are now in the specified order.

```
ans =
    'height'
```

```
'dhdt'
'Current'
```

Validate Linearization In Time Domain

In this section...

Validate Linearization in Time Domain

This example shows how to validate linearization results by comparing the simulated output of the nonlinear model and the linearized model.

1 Linearize Simulink model.

For example:

```
sys = 'watertank';
load_system(sys);
sys_io(1) = linio('watertank/PID Controller',1,'input');
sys_io(2) = linio('watertank/Water-Tank System',1,'openoutput');
opspec = operspec(sys);
op = findop(sys,opspec,findopOptions('DisplayReport','off'));
linsys = linearize(sys,op,sys_io);
```

If you linearized your model in the Linear Analysis Tool, you must export the linear model to the MATLAB workspace.

2 Create input signal for validation. For example, a step input signal:

```
input = frest.createStep('Ts',0.1,...
                         'StepTime',1,...
                         'StepSize',1e-5,...
                         'FinalTime',500);
```

3 Simulate the Simulink model using the input signal.

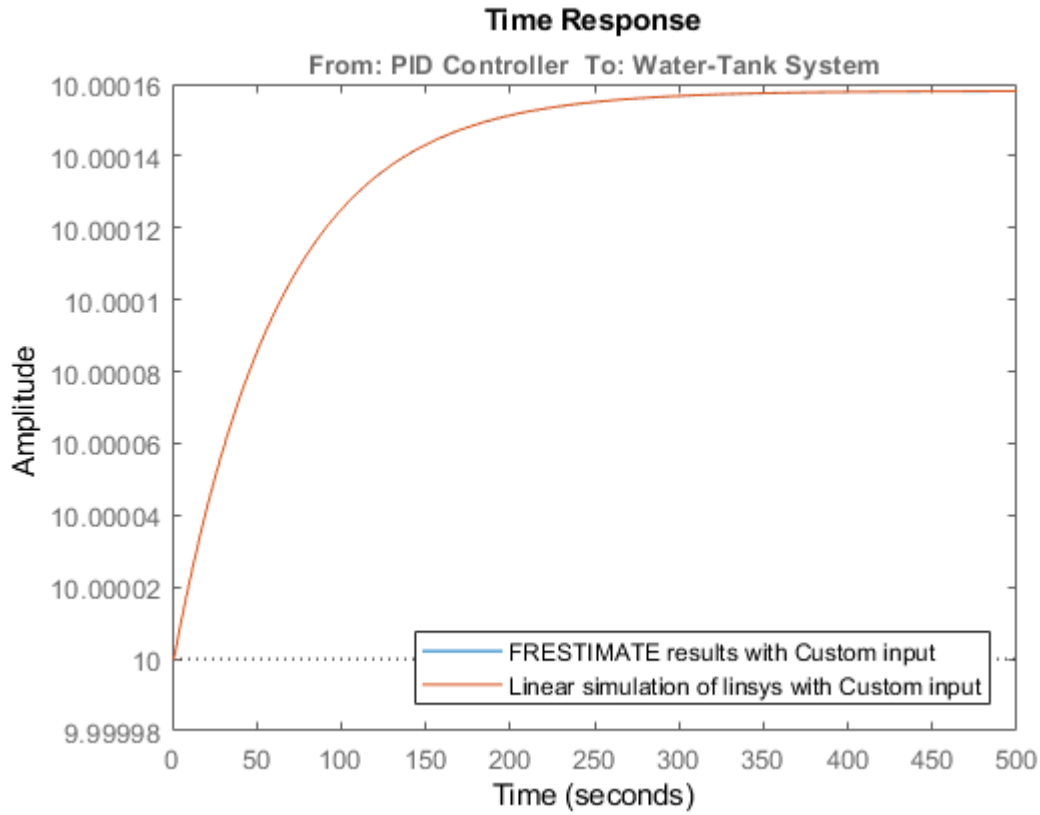
```
[~,simout] = frestimate(sys,op,sys_io,input);
```

`simout` is the simulated output of the nonlinear model.

4 Simulate the linear model `sys`, and compare the time-domain responses of the linear and nonlinear Simulink model.

```
frest.simCompare(simout,linsys,input)
legend('FRESTIMATE results with Custom input',...
```

```
'Linear simulation of linsys with Custom input',...
'Location','SouthEast');
```



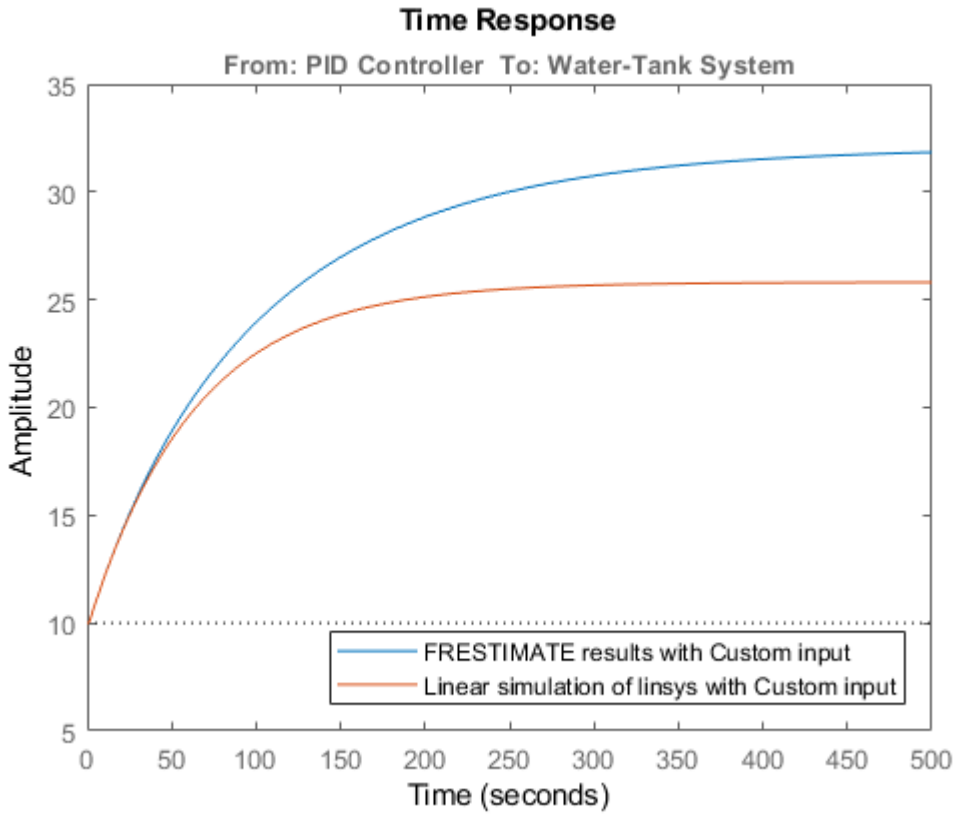
The step response of the nonlinear model and linearized model are close, which validates that the linearization is accurate.

5 Increase the amplitude of the step signal from `1.0e-005` to `1`.

```
input = frest.createStep('Ts',0.1,...
                         'StepTime',1,...
                         'StepSize',1,...
                         'FinalTime',500);
```

6 Repeat the frequency response estimation with the increased amplitude of the input signal, and compare this time response plot to the exact linearization results.

```
[~,simout2] = frestimate(sys,op,sys_io,input);
frest.simCompare(simout2,linsys,input)
legend('FRESTIMATE results with Custom input',...
            'Linear simulation of linsys with Custom input',...
            'Location','SouthEast');
```



The step response of linear system you obtained using exact linearization does not match the step response of the estimated frequency response with large input signal amplitude. The linear model obtained using exact linearization does not match the full nonlinear model at amplitudes large enough to deviate from the specified operating point.

Choosing Time-Domain Validation Input Signal

For time-domain validation of linearization, use `frest.createStep` to create a step signal. Use the step signal as an input to `frest.simCompare`, which compares the simulated output of the nonlinear model and the linearized model.

The step input helps you assess whether the linear model accurately captures the dominant time constants as it goes through the step transients.

The step input also shows whether you correctly captured the DC gain of the Simulink model by comparing the final value of the exact linearization simulation with the frequency response estimation.

Validate Linearization In Frequency Domain

In this section...

Validate Linearization in Frequency Domain using Linear Analysis Tool

This example shows how to validate linearization results using an estimated linear model.

In this example, you linearize a Simulink model using the I/Os specified in the model. You then estimate the frequency response of the model using the same operating point (model initial condition). Finally, you compare the estimated response to the exact linearization result.

Linearize Simulink Model

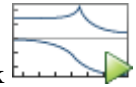
1 Open the model.

```
sys = 'scdDCMotor';
open_system(sys)
```

2 Open the Linear Analysis Tool for the model.

In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

3 Linearize the model at the default operating point and analysis I/Os, and generate a bode plot of the result.

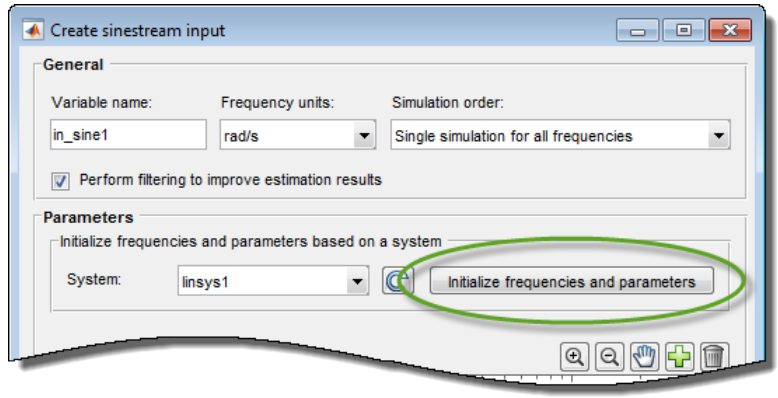
Click  **Bode**. The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the Linear Analysis Workspace.

Estimate Frequency Response of Model

1 Create a sinestream input signal for computing an approximation of the model by frequency response estimation. In the **Estimation** tab, in the **Input Signal** drop-down list, select `Sinestream`.

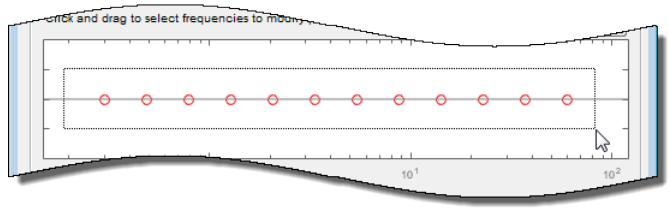
2 Initialize the input signal frequencies and parameters based on the linearized model.

Click **Initialize frequencies and parameters**.

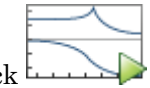


The frequency display in the dialog box is populated with frequency points. The software chooses the frequencies and input signal parameters automatically based on the dynamics of `linsys1`.

3 Set the amplitude of the input signal at all frequency points to 1. In the frequency display, select all the frequency points.

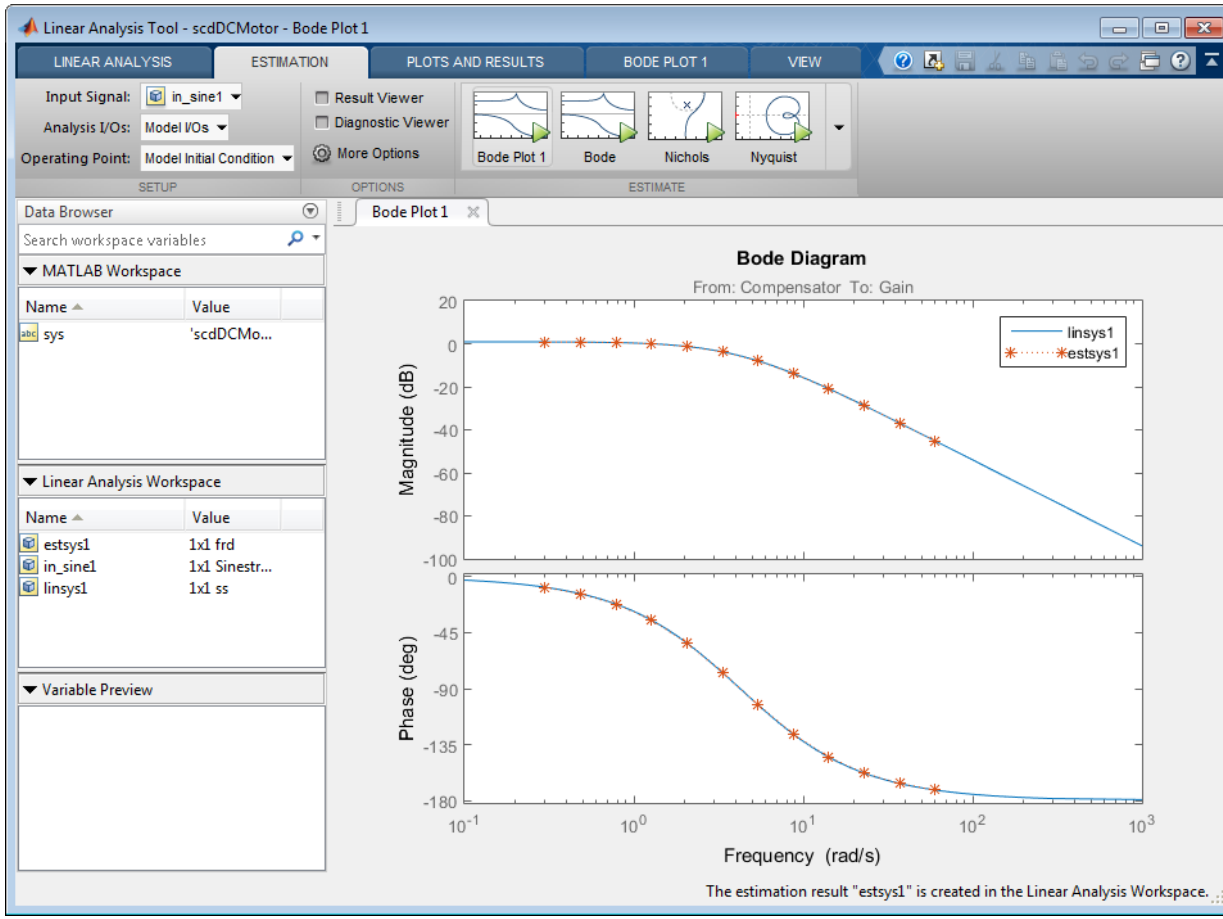


Enter 1 in the **Amplitude** field, and click **OK**. The new input signal `in_sine1` appears in the **Linear Analysis Workspace**.

4 Estimate the frequency response and plot its frequency response on the existing Bode plot of the linearized system response. Click  **Bode Plot 1**.

Examine estimation results.

Bode Plot 1 now shows the Bode responses for the estimated model and the linearized model.



The frequency response for the estimated model matches that of the linearized model.

For more information about frequency response estimation, see “What Is a Frequency Response Model?” on page 5-2.

Choosing Frequency-Domain Validation Input Signal

For frequency-domain validation of linearization, create a sinestream signal. By analyzing one sinusoidal frequency at a time, the software can ignore some of the impact of nonlinear effects.

Input Signal	Use When	See Also
Sinestream	All linearization inputs and outputs are on continuous signals.	`frest.Sinestream`
Sinestream with fixed sample time	One or more of the linearization inputs and outputs is on a discrete signal	`frest.createFixedTsSinestream`

You can easily create a sinestream signal based on your linearized model. The software uses the linearized model characteristics to accurately predict the number of sinusoid cycles at each frequency to reach steady state.

When diagnosing the frequency response estimation, you can use the sinestream signal to determine whether the time series at each frequency reaches steady state.

See Also

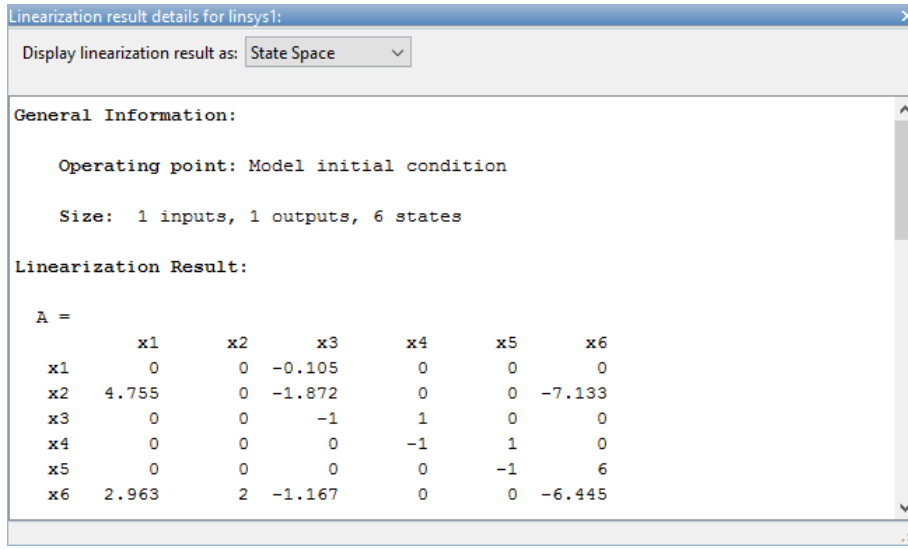
More About

- “Estimation Input Signals” on page 5-7

View Linearized Model Equations Using Linear Analysis Tool

When you linearize a Simulink model using the Linear Analysis Tool, the software generates state-space equations for the resulting linear model. To view the linearized model equations:

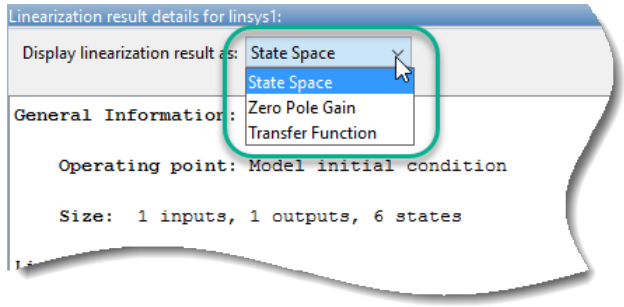
1. In the **Data Browser**, in the **Linear Analysis Workspace**, select the linear model you want to view.
2. On the **Plots and Results** tab, click  **Result Viewer**.



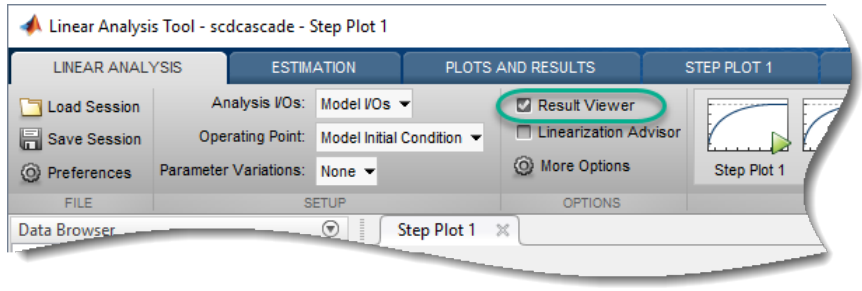
In the Linearization result details dialog box, the software displays:

- General information about the linearization, including the operating point and the number of inputs, outputs, and states.
- State-space matrices for the linearized model.
- Lists of the state, input, and output names. To highlight a state, input, or output in the Simulink model, click the corresponding name.

To display the system using either zero-pole-gain or transfer function equations, in the **Display linearization result as** drop-down list, select a format.



You can automatically open the Linearization result details dialog box when you linearize your model. To do so, on the **Linear Analysis** tab, select **Result Viewer** before you linearize the model.



See Also

Apps

Linear Analysis Tool

More About

- “Analyze Results Using Linear Analysis Tool Response Plots” on page 2-146

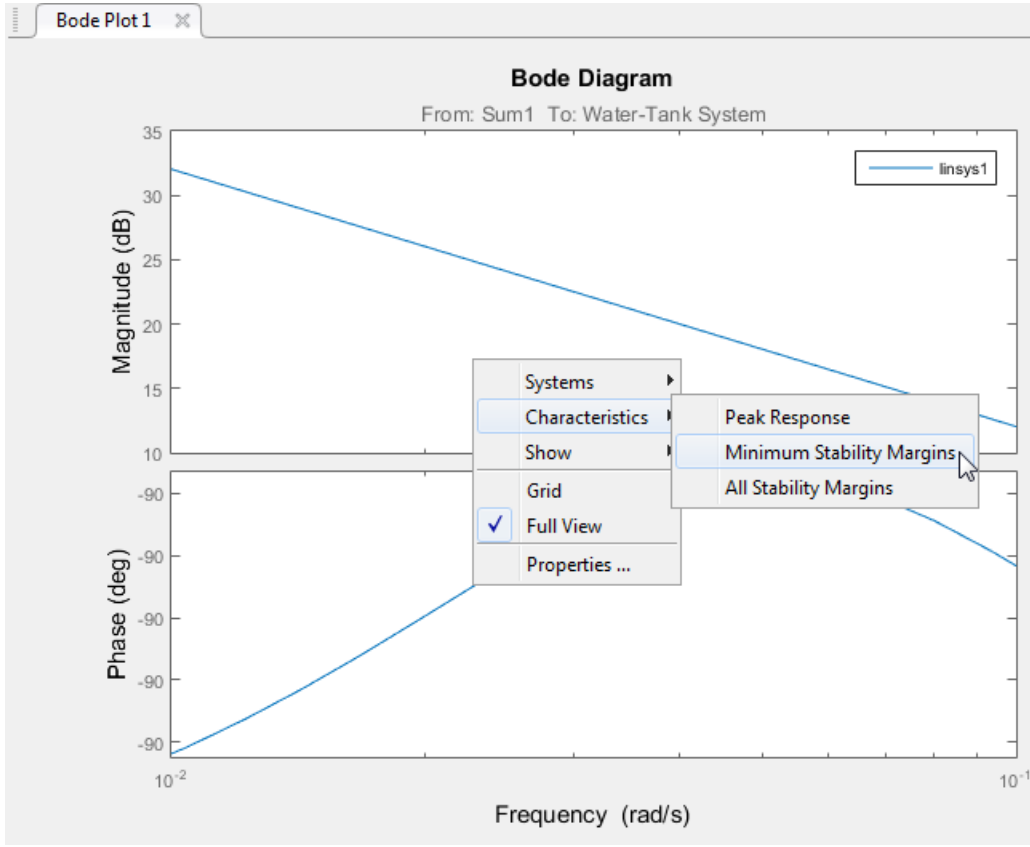
Analyze Results Using Linear Analysis Tool Response Plots

This topic explains ways to use and manipulate response plots of linearized systems in Linear Analysis Tool.

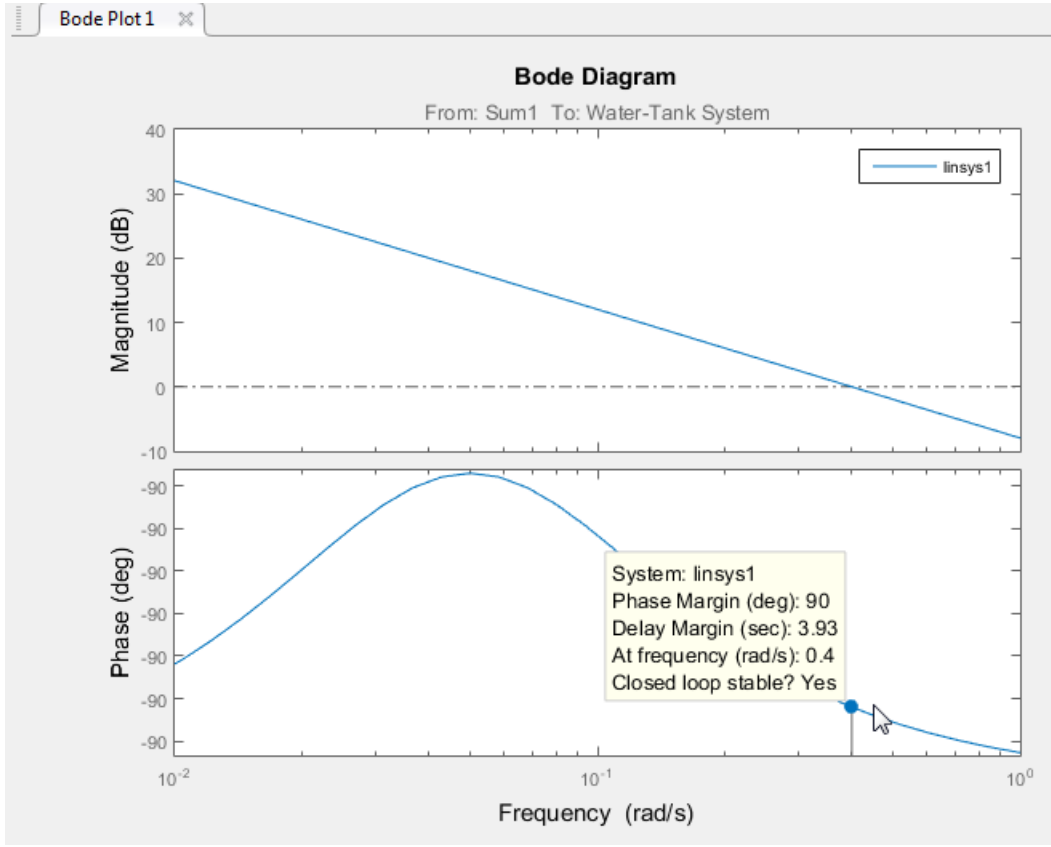
In this section...

View System Characteristics on Response Plots

To view system characteristics such as stability margins, overshoot, or settling time on a Linear Analysis Tool response plot, right-click the plot and select **Characteristics**. Then select the system characteristic you want to view.



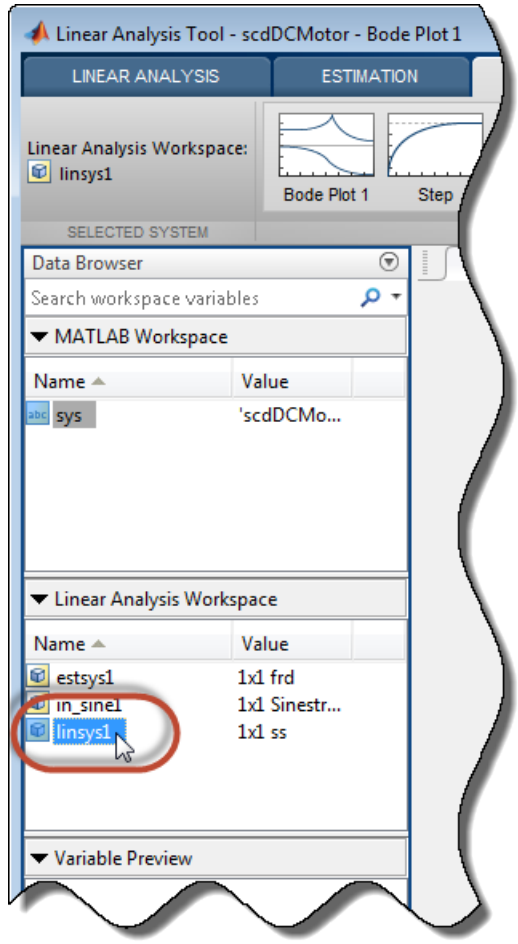
For most characteristics, a data marker appears on the plot. Click the marker to show a data tip that contains information about the system characteristic.



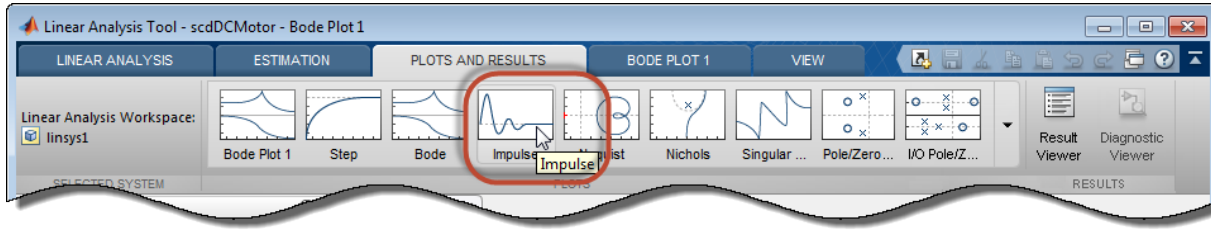
Generate Additional Response Plots of Linearized System

In Linear Analysis Tool, when you have linearized or estimated a system, generate additional response plots of the system as follows:

1 In the Linear Analysis Tool, click the **Plots and Results** tab. In the Linear Analysis Workspace or the MATLAB Workspace, select the system you want to plot.

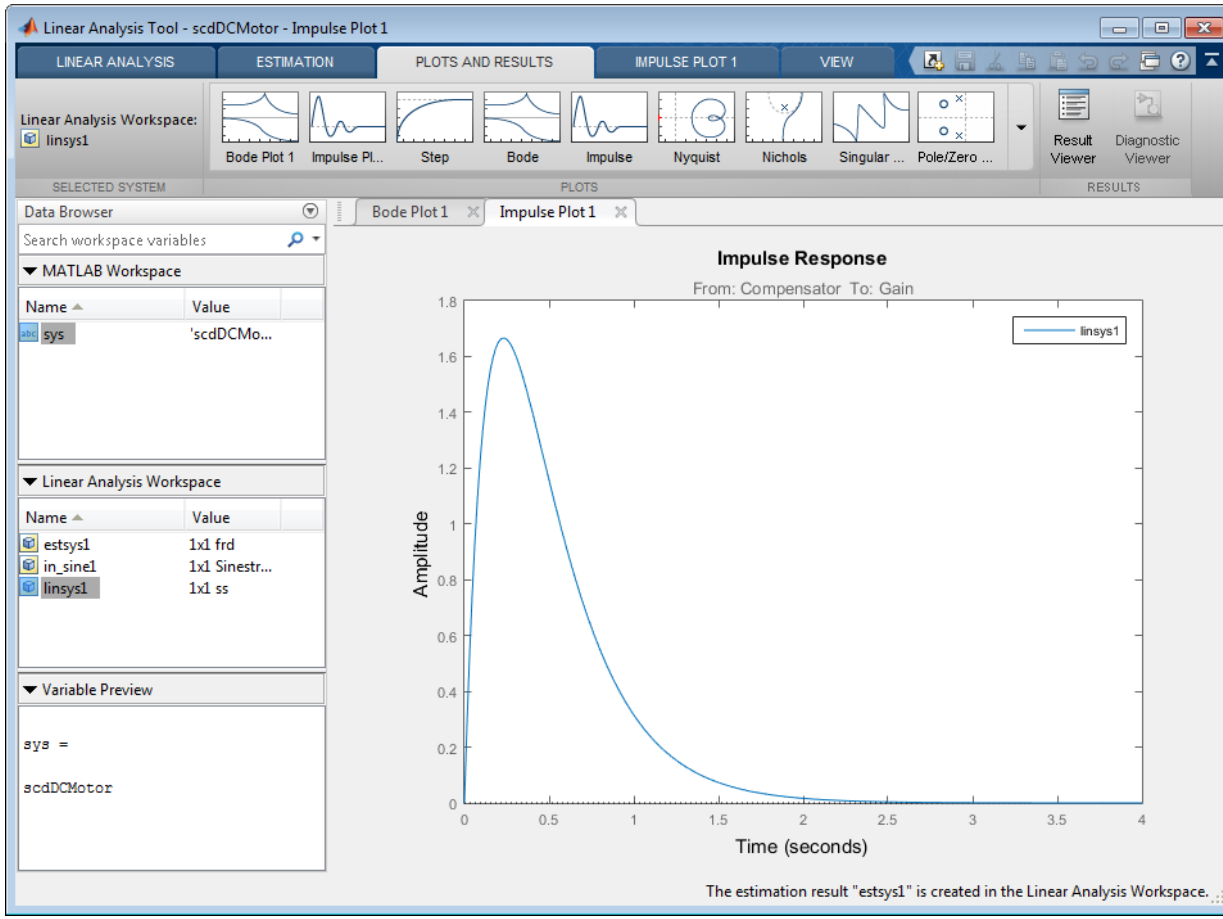


2 In the **Plots** section of the tab, click the type of plot you want to generate.



Tip Click ▾ to expand the gallery view.

Linear Analysis Tool generates a new plot of type you select.

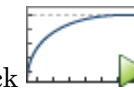


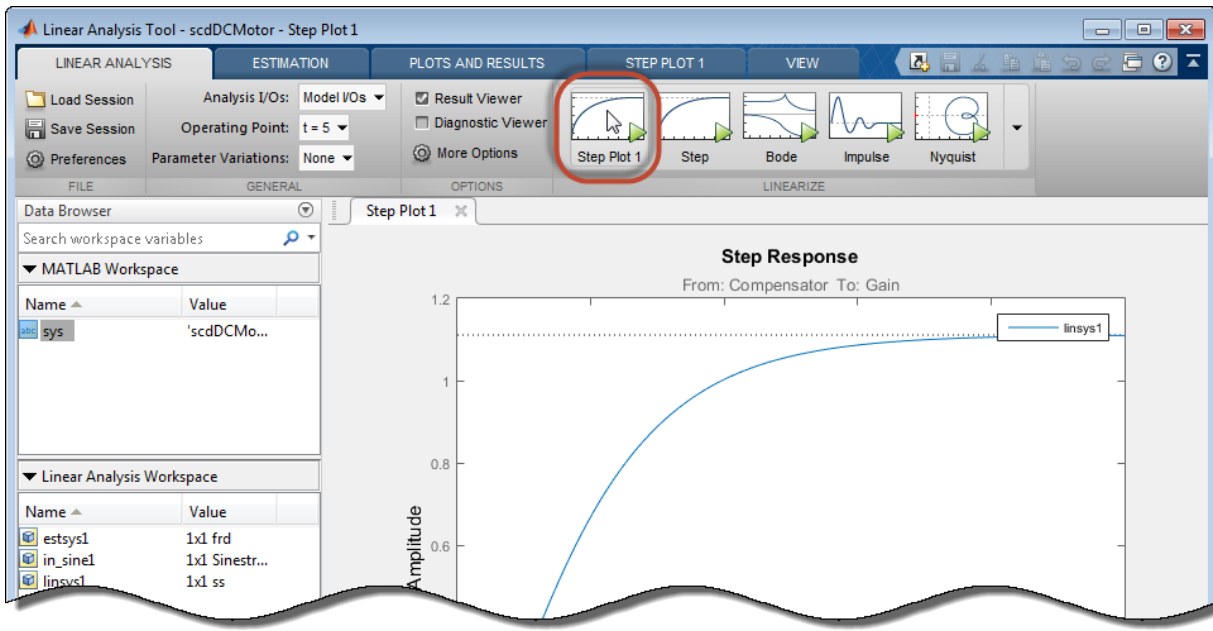
Tip To multiple plots at the same time, select a layout in the **View** tab.

Add Linear System to Existing Response Plot

New Linear System

When you compute a new linearization or frequency response estimation, in the **Linear Analysis** tab, click the button corresponding to an existing plot to add the new linear system to that plot.

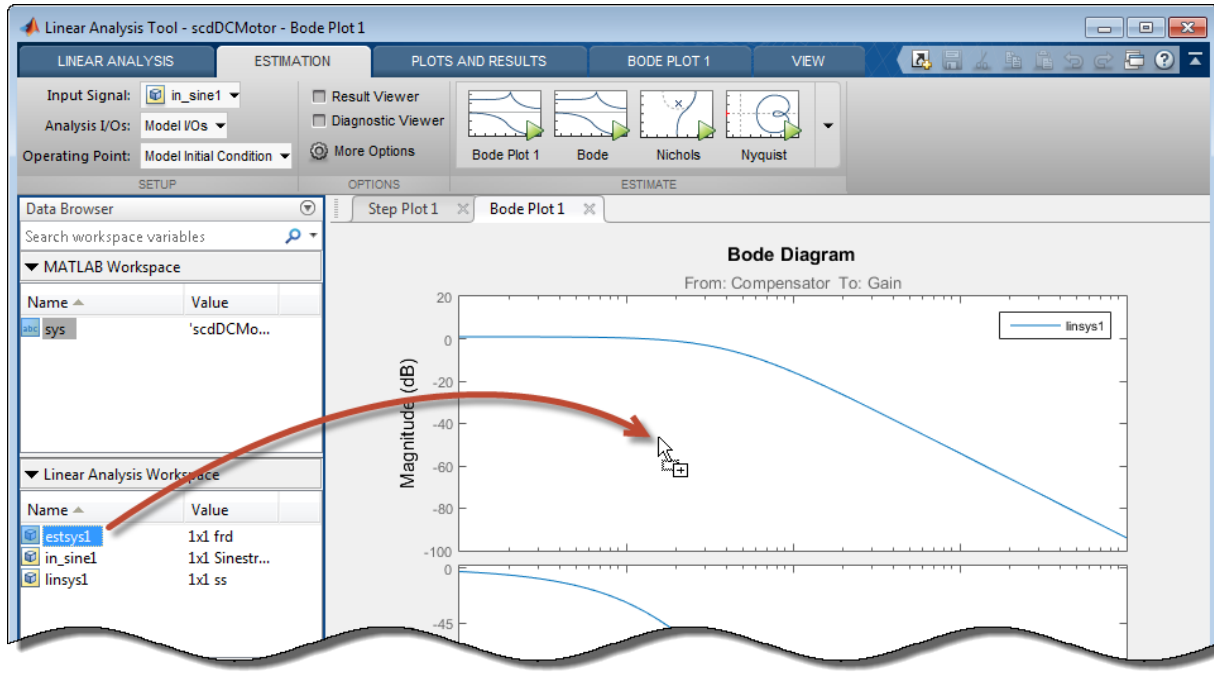
For example, suppose that you have linearized a model at the default operating point for the model, and have a step plot of the result, `Step Plot 1`. Suppose further that you have specified a new operating point, a linearization snapshot time. To linearize at the new operating point and add the result to `Step Plot 1`, click  **Step Plot 1**. Linear Analysis Tool computes the new linearization and adds the step response of the new system, `linsys2`, to the existing step response plot.



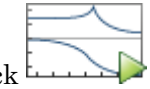
Linear System in Workspace

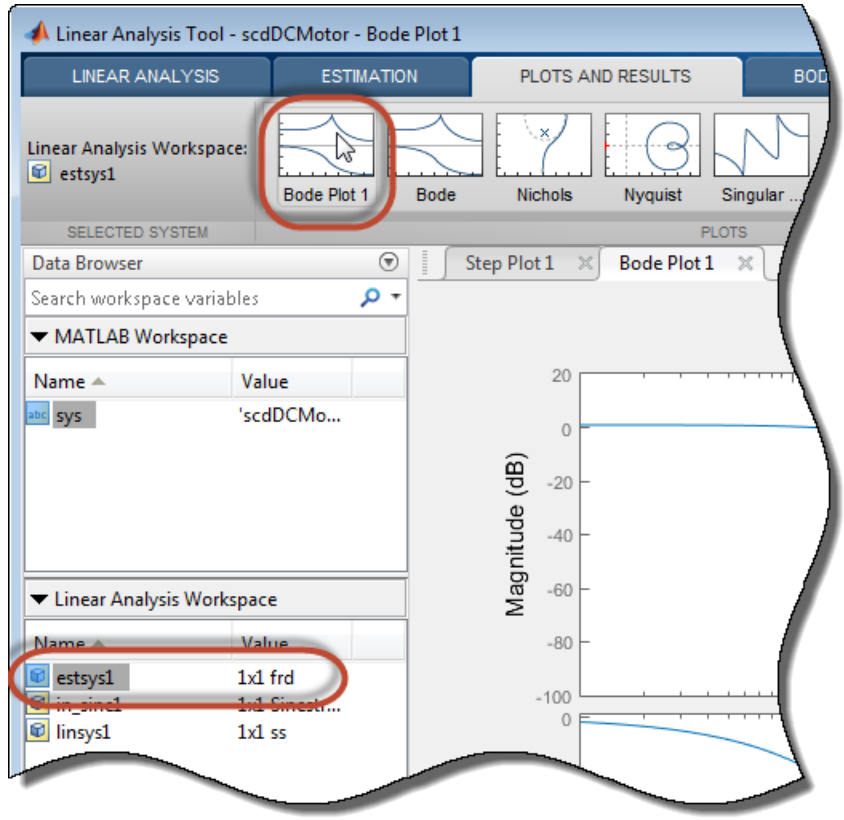
There are two ways to add a linear system from the MATLAB Workspace or the Linear Analysis Workspace to an existing plot in the Linear Analysis Tool.

- Drag the linear system onto the plot from the MATLAB Workspace or the Linear Analysis Workspace.



- On the **Plots and Results** tab, in the Linear Analysis Workspace, select the system you want to add to an existing plot. Then, in the **Plots** section of the tab, select the button corresponding to the existing plot you want to update.

  For example, suppose that you have a Bode plot of the response of a linear system, `Bode Plot 1`. Suppose further that you have an estimated response in the Linear Analysis Workspace, `estsys1`. To add the response of `estsys1` to the existing Bode plot, select `estsys1` and click  **Bode Plot 1**.



Tip Click ▾ to expand the gallery view.

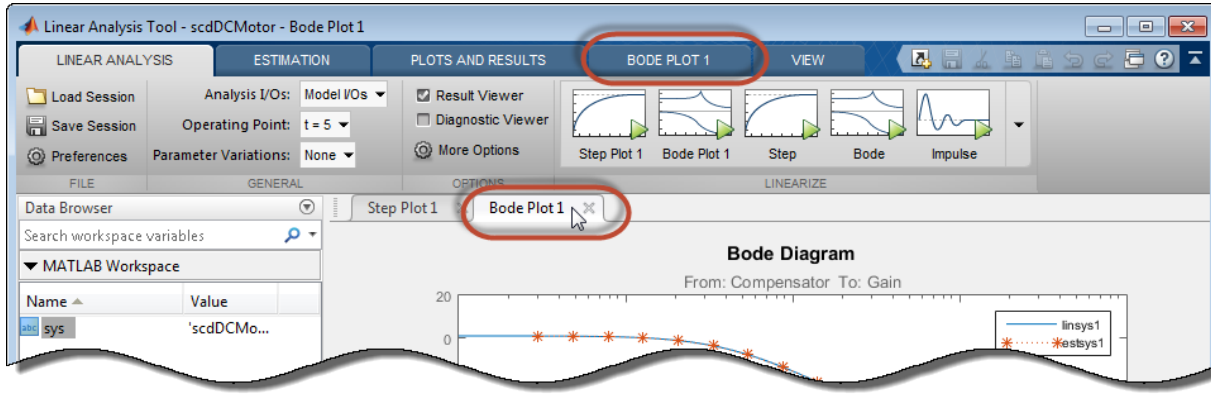
Customize Characteristics of Plot in Linear Analysis Tool

To change the characteristics of an existing plot, such as the title, axis labels, or text styles, double-click the plot to open the properties editor. Edit plot properties as desired. Plots are updated as you make changes. Click **Close** when you are finished.

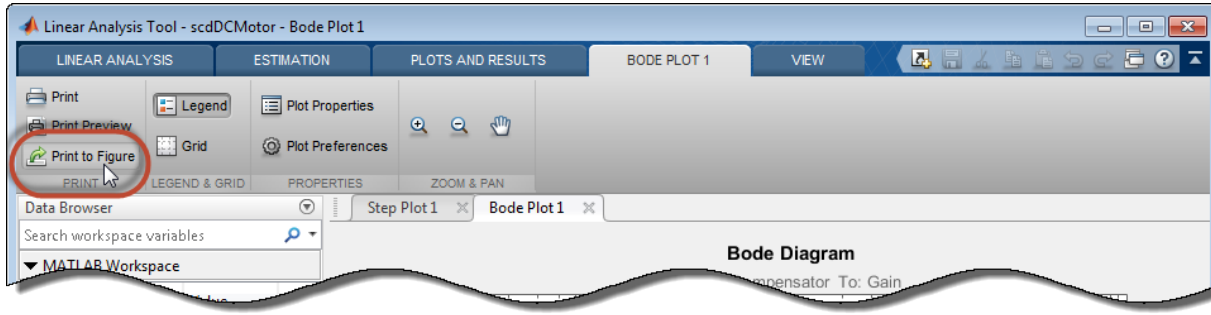
Print Plot to MATLAB Figure in Linear Analysis Tool

To export a plot from the Linear Analysis Tool to a MATLAB figure window:

1 Select the plot you want to export. A tab appears with the same name as the plot.



2 Click the new tab. In the **Print** section, click **Print to Figure**.



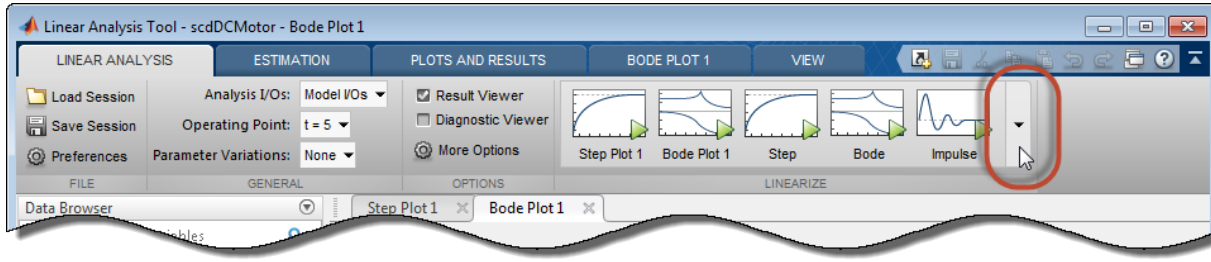
A MATLAB figure window opens containing the plot.

Generate MATLAB Code for Linearization from Linear Analysis Tool

This topic shows how to generate MATLAB code for linearization from the Linear Analysis Tool. You can generate either a MATLAB script or a MATLAB function. Generated MATLAB scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB function allows you to perform multiple linearizations with systematic variations in linearization parameters such as operating point (batch linearization).

To generate MATLAB code for linearization:

1. In the Linear Analysis Tool, in the **Linear Analysis** tab, interactively configure the analysis I/Os, operating point, and other parameters for linearization.
2. Click ▼ to expand the gallery.



3. In the gallery, click the button for the type of code you want to generate:
   - `Script` — Generate a MATLAB script that uses your configured parameter values and operating point. Select this option when you want to repeat the same linearization at the MATLAB command line.
   - `Function` — Generate a MATLAB function that takes analysis I/Os and operating points as input arguments. Select this option when you want to perform multiple linearizations using different parameter values (batch linearization).

See Also

`linearize`

More About

- “What Is Batch Linearization?” on page 3-2
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-28

When to Specify Individual Block Linearization

Some Simulink blocks, including those with sharp discontinuities, can produce poor linearization results. For example, when your model operates in a region away from the point of discontinuity, the linearization of the block is zero. Typically, you must specify custom linearizations for such blocks. You can specify the block linearization as:

- A linear model in the form of a D-matrix.
- A Control System Toolbox model object.
- An uncertain state-space object or an uncertain real object (requires Robust Control Toolbox software).

See Also

More About

- "Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-158
- "Specify D-Matrix System for Block Linearization Using Function" on page 2-159
- "Augment the Linearization of a Block" on page 2-163
- "Change Perturbation Level of Blocks Perturbed During Linearization" on page 2-181
- "Block Linearization Troubleshooting" on page 4-58

Specify Linear System for Block Linearization Using MATLAB Expression

This example shows how to specify the linearization of any block, subsystem, or model reference without having to replace this block in your Simulink model.

1. Right-click the block in the model, and select **Linear Analysis > Specify Selected Block Linearization**.

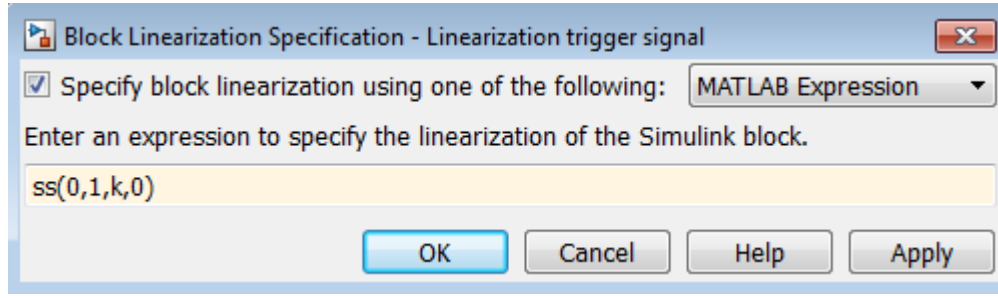
   The Block Linearization Specification dialog box opens.

2. In the **Specify block linearization using one of the following** list, select **MATLAB Expression**.

3. In the text field, enter an expression that specifies the linearization.

   For example, specify the linearization as an integrator with a gain of k, $G(s) = k/s$.

   In state-space form, this transfer function corresponds to `ss(0,1,k,0)`.

   Click **OK**.

4. Linearize the model.

See Also

Related Examples

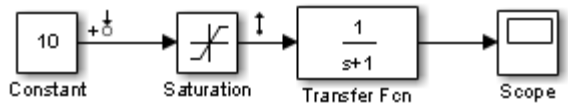
- "Specify D-Matrix System for Block Linearization Using Function" on page 2-159
- "Block Linearization Troubleshooting" on page 4-58

Specify D-Matrix System for Block Linearization Using Function

This example shows how to specify custom linearization for a saturation block using a function.

1 Open Simulink model.

```
sys = 'configSatBlockFcn';
open_system(sys)
```



In this model, the limits of the saturation block are `-satlimit` and `satlimit`. The current value of the workspace variable `satlimit` is 10.

2 Linearize the model at the model operating point using the linear analysis points defined in the model. Doing so returns the linearization of the saturation block.

```
io = getlinio(sys);
linsys = linearize(sys,io)
```

```
linsys =

  D =
                Constant
   Saturation          1

Static gain.
```

At the model operating point, the input to the saturation block is 10. This value is right on the saturation boundary. At this value, the saturation block linearizes to 1.

3 Suppose that you want the block to linearize to a transitional value of 0.5 when the input falls on the saturation boundary. Write a function that defines the saturation block linearization to behave this way. Save the function to the MATLAB path.

```
function blocklin = mySaturationLinearizationFcn(BlockData)
% This function customizes the linearization of a saturation block
% based on the block input signal level, U:
% BLOCKLIN = 0 when |U| > saturation limit
```

```
% BLOCKLIN = 1 when |U| < saturation limit
% BLOCKLIN = 1/2 when U = saturation limit

% Get saturation limit.
satlimit = BlockData.Parameters.Value;

% Compute linearization based on the input signal
% level to the block.
if abs(BlockData.Inputs(1).Values) > satlimit
    blocklin = 0;
elseif abs(BlockData.Inputs(1).Values) < satlimit
    blocklin = 1;
else
    blocklin = 1/2;
end
```

This configuration function defines the saturation block linearization based on the level of the block input signal. For input values outside the saturation limits, the block linearizes to zero. Inside the limits, the block linearizes to 1. Right on the boundary values, the block linearizes to the interpolated value of 0.5. The input to the function, `BlockData`, is a structure that the software creates automatically when you configure the linearization of the Saturation block to use the function. The configuration function reads the saturation limits from that data structure.

4 In the Simulink model, right-click the Saturation block, and select **Linear Analysis > Specify Selected Block Linearization**.

The Block Linearization Specification dialog box opens.

5 Check **Specify block linearization using one of the following**. Choose **Configuration Function** from the list.

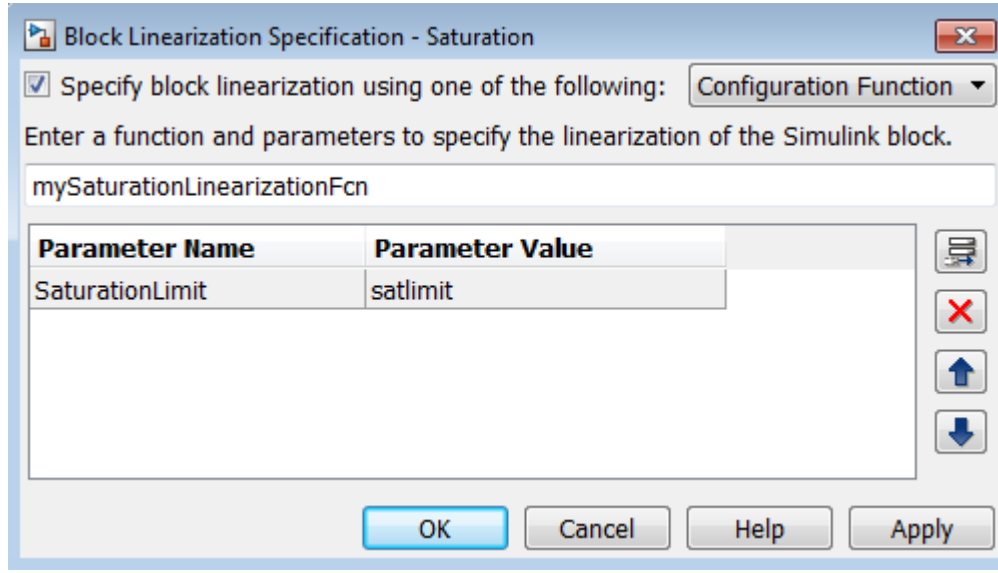
Configure the linearization function:

a Enter the name you gave to your saturation function. In this example, the function name is `mySaturationLinearizationFcn`.

b Specify the function parameters. `mySaturationLinearizationFcn` requires the saturation limit value, which the user must specify before linearization.

Enter the variable name `satlimit` in **Parameter Value**. Enter the corresponding descriptive name in the **Parameter Name** column, `SaturationLimit`.

c Click **OK**.



Configuring the Block Linearization Specification dialog box updates the model to use the specified linearization function for linearizing the Saturation Block. Specifically, this configuration automatically populates the `Parameters` field of the `BlockData` structure, which is the input argument to the configuration function.

Note You can add function parameters by clicking . Use to delete selected parameters.

Code Alternative

This code is equivalent to configuring the Block Linearization Specification dialog box:

```
satblk = 'configSatBlockFcn/Saturation';
set_param(satblk,'SCDEnableBlockLinearizationSpecification','on')
rep = struct('Specification','mySaturationLinearizationFcn',...
             'Type','Function',...
             'ParameterNames','SaturationLimit',...
```

```
                    'ParameterValues','satlimit');
set_param(satblk,'SCDBlockLinearizationSpecification',rep)
```

6 Define the saturation limit, which is a parameter required by the linearization function of the Saturation block.

```
satlimit = 10;
```

7 Linearize the model again. Now, the linearization uses the custom linearization of the saturation block.

```
linsys_cust = linearize(sys,io)

linsys_cust =

  d =
                Constant
   Saturation        0.5

Static gain.
```

At the model operating point, the input to the saturation block is 10. Therefore, the block linearizes to 0.5, the linearization value specified in the function for saturation boundary.

See Also

More About

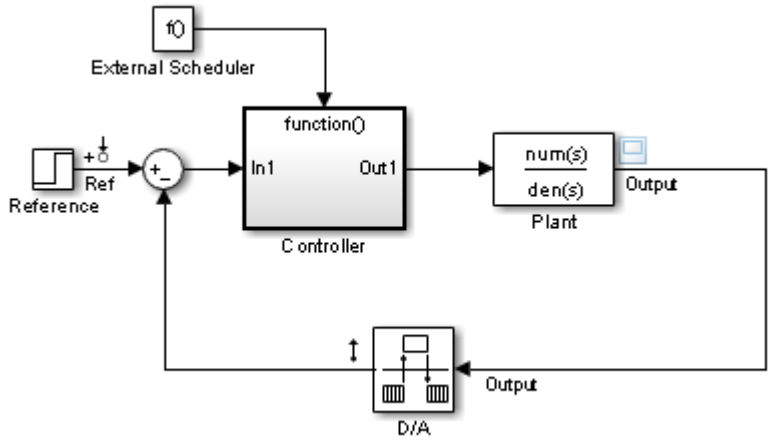
- “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-158
- “Block Linearization Troubleshooting” on page 4-58

Augment the Linearization of a Block

This example shows how to augment the linearization of a block with additional time delay dynamics, using a block linearization specification function.

1 Open Simulink model.

```
mdl = 'scdFcnCall';
open_system(mdl)
```



This model includes a continuous time plant, `Plant`, and a discrete-time controller, `Controller`. The `D/A` block discretizes the plant output with a sampling time of 0.1 s. The `External Scheduler` block triggers the controller to execute with the same period, 0.1 s. However, the trigger has an offset of 0.05 s relative to the discretized plant output. For that reason, the controller does not process a change in the reference signal until 0.05 s after the change occurs. This offset introduces a time delay of 0.05 s into the model.

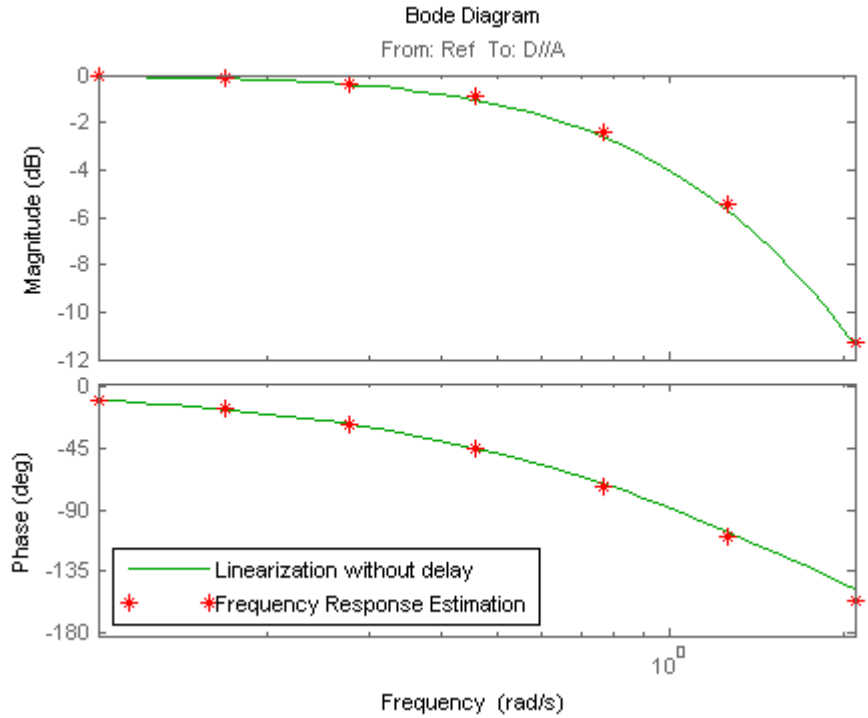
2 (Optional) Linearize the closed-loop model at the model operating point without specifying a linearization for the `Controller` block.

```
io = getlinio(mdl);
sys_nd = linearize(mdl,io);
```

The `getlinio` function returns the linearization input and output points that are already defined in the model.

3 (Optional) Check the linearization result by frequency response estimation.

```
input = frest.Sinestream(sys_nd);
sysest = frestimate(mdl,io,input);
bode(sys_nd,'g',sysest,'r*',{input.Frequency(1),input.Frequency(end)})
legend('Linearization without delay',...
     'Frequency Response Estimation','Location','SouthWest')
```



The exact linearization does not account for the time delay introduced by the controller execution offset. A discrepancy results between the linearized model and the estimated model, especially at higher frequencies.

4 Write a function to specify the linearization of the `Controller` block that includes the time delay.

The following configuration function defines a linear system that equals the default block linearization multiplied by a time delay. Save this configuration function to a

location on your MATLAB path. (For this example, the function is already saved as `scdAddDelayFcn.m`.)

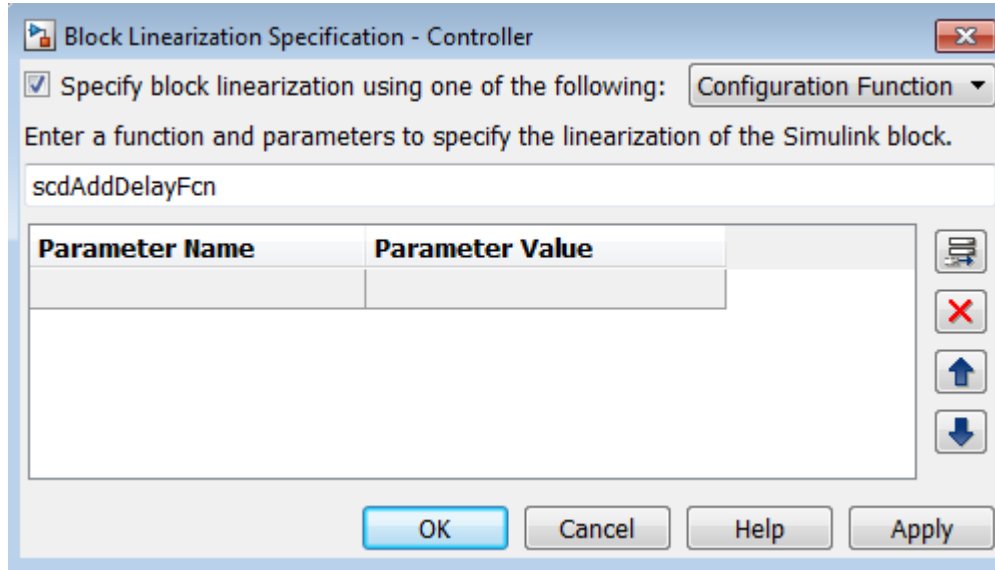
```
function sys = scdAddDelayFcn(BlockData)
sys = BlockData.BlockLinearization*thiran(0.05,0.1);
```

The input to the function, `BlockData`, is a structure that the software creates automatically each time it linearizes the block. When you specify a block linearization configuration function, the software automatically passes `BlockData` to the function. The field `BlockData.BlockLinearization` contains the current linearization of the block.

This configuration function approximates the time delay as a `thiran` filter. The filter indicates a discrete-time approximation of the fractional time delay of 0.5 sampling periods. (The 0.05 s delay has a sampling time of 0.1 s).

5 Specify the configuration function `scdAddDelayFcn` as the linearization for the `Controller` block.

- **a** Right-click the `Controller` block, and select **Linear Analysis > Specify Selected Block Linearization**.
- **b** Select the **Specify block linearization using one of the following** check box. Then, select **Configuration Function** from the drop-down list.
- **c** Enter the function name `scdAddDelayFcn` in the text box. `scdAddDelayFcn` has no parameters, so leave the parameter table blank.
- **d** Click **OK**.



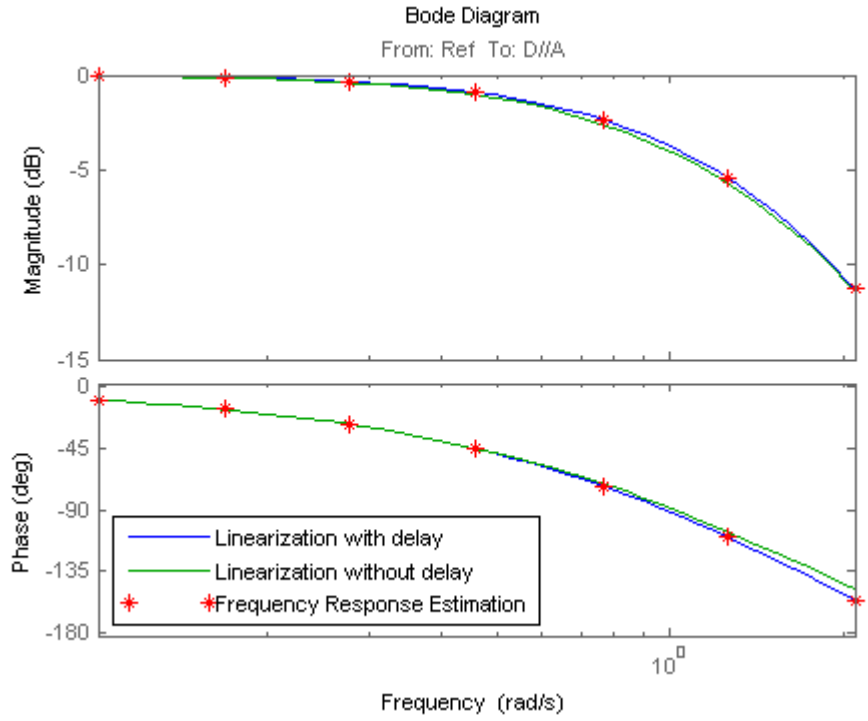
6 Linearize the model using the specified block linearization.

```
sys_d = linearize(mdl,io);
```

The linear model `sys_d` is a linearization of the closed-loop model that accounts for the time delay.

7 (Optional) Compare the linearization that includes the delay with the estimated frequency response.

```
bode(sys_d,'b',sys_nd,'g',sysest,'r*',...
     {input.Frequency(1),input.Frequency(end)})
legend('Linearization with delay','Linearization without delay',...
     'Frequency Response Estimation','Location','SouthWest')
```



The linear model obtained with the specified block linearization now accounts for the time delay. This linear model is therefore a much better match to the real frequency response of the Simulink model.

See Also

`getlinio` | `linearize`

Models with Time Delays

In this section...

Choose Approximate Versus Exact 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 Exact Representation of Time Delays

Before linearizing your model:

- In the Linear Analysis Tool:

  1. On the **Linear Analysis** tab, click **More Options**.
  2. In the Options for exact linearization dialog box, in the **Linearization** tab, check **Return linear model with exact delay(s)**.

- At the command line, create a `linearizeOptions` option set, setting the `UseExactDelayModel` to `'on'`.

See Also

`linearizeOptions`

More About

- "Time Delays in Linear Systems" (Control System Toolbox)
- "Time-Delay Approximation" (Control System Toolbox)
- "Linearization of Models with Delays"

Linearize Multirate Models

You can linearize a Simulink model that contains blocks with different sample times using Simulink Control Design software. By default, the linearization tools:

- Convert sample times using a zero-order hold conversion method.
- Create a linearized model with a sample time equal to the largest sample time of the blocks on the linearization path.

You can change either of these behaviors by specifying linearization options, which affects the linearization result.

Change Sample Time of Linear Model

By default, the software chooses the largest sample time of the multirate model. If the default sample time is not appropriate for your application, you can specify a different sample time.

To specify the sample time of the linear model in the Linear Analysis Tool:

1. On the **Linear Analysis** tab, click **More Options**.
2. In the Options for exact linearization dialog box, on the **Linearization** tab, in the **Enter sample time (sec)** field, specify the sample time. You can specify any of the following values:
   - `-1` — Use the largest sample time from the model.
   - `0` — Create a continuous-time model. In this case, the software creates a discrete-time model using the largest sample time from the model, then converts the resulting model to continuous time.
   - Positive scalar — Use the specified value for the sample time.

To specify the sample time of the linear model at the command line, create a `linearizeOptions` option set, and set the `SampleTime` option. For example:

```
opt = linearizeOptions;
opt.SampleTime = 0.01;
```

You can then use this option set with `linearize` or `slLinearizer`.

Change Linearization Rate Conversion Method

When you linearize models with multiple sample times, such as a discrete controller with a continuous plant, the software uses a rate conversion algorithm to create a single-rate linear model. The default rate conversion method is zero-order hold.

To specify the rate conversion method in the Linear Analysis Tool:

1. On the **Linear Analysis** tab, click  **More Options**.
2. In the Options for exact linearization dialog box, on the **Linearization** tab, in the **Choose rate conversion method** drop-down list, select one of the following rate conversion methods.

Rate Conversion Method	When to Use
`Zero-Order Hold`	You need exact discretization of continuous dynamics in the time domain for staircase inputs.
`Tustin`	You need good frequency-domain matching between a continuous-time system and the corresponding discretized system, or between an original system and a resampled system.
`Tustin with Prewarping`	You need good frequency-domain matching at a particular frequency between a continuous-time system and the corresponding discretized system, or between an original system and the resampled system.

`Upsampling when possible, Zero-Order Hold otherwise`
`Upsampling when possible, Tustin otherwise`
`Upsampling when possible, Tustin with Prewarping otherwise`
	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 the alternate rate conversion method.

3. If you select either of the following rate conversion methods:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

then, in the **Enter prewarp frequency** field, specify the prewarp frequency.

To specify the rate conversion method at the command line, create a `linearizeOptions` option set, and set the `RateConversionMethod` and `PreWarpFreq` options. For example:

```
opt = linearizeOptions;
opt.RateConversionMethod = 'prewarp';
opt.PreWarpFreq = 100;
```

You can then use this option set with `linearize` or `slLinearizer`.

Note If you use a rate conversion method other than zero-order hold, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system are `'?'`.

Linearization Using Different Rate Conversion Methods

This example shows how to specify the rate conversion method for the linearization of a multirate model. The choice of rate conversion methodology can affect the resulting linearized model. This example illustrates the extraction of a discrete linear time invariant model using two different rate conversion methods.

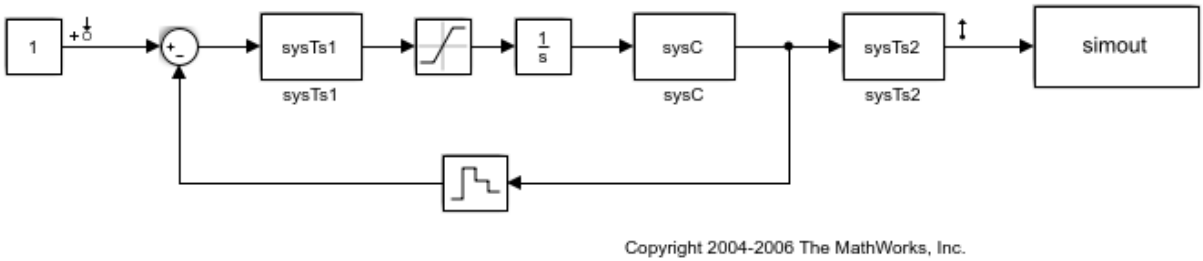
Example Problem

In the Simulink model `scdmrate.mdl` there are three different sample rates specified in five blocks. These blocks are

- `sysC` - a continuous linear block,
- `Integrator` - a continuous integrator,
- `sysTs1` - a block that has a sample time of 0.01 seconds,
- `sysTs2` - a block that has a sample time of 0.025 seconds, and
- `Zero-Order Hold` - a block that samples the incoming signal at 0.01 seconds.

Open the Simulink model.

scdmrate

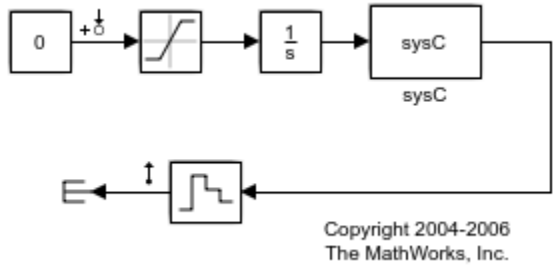


In this example, you linearize the model between the output of the block `sysTs1` and the block `Zero-Order Hold`. Additionally, you add a loop opening at the block `Zero-Order Hold` to extract the plant model for the system.

```
model = 'scdmrate';
io(1) = linio('scdmrate/sysTs1',1,'input');
io(2) = linio('scdmrate/Zero-Order Hold',1,'openoutput');
```

Using these linearization points the linearization effectively results in the linearization of the model `scdmrate_ol`.

scdmrate_ol



When linearizing a model that contains both continuous and discrete signals, the software first converts the continuous signals to discrete signals, using a rate conversion method. The default rate conversion method is zero-order hold. To view or change the rate conversion method, use the `RateConversionMethod` property in the linearizeOptions function. The following command shows that `RateConversionMethod` is set to the default setting, `zoh`:

```
opt = linearizeOptions


Options for LINEARIZE:
    LinearizationAlgorithm         : blockbyblock
    SampleTime (-1 Auto Detect)    : -1
    UseFullBlockNameLabels (on/off): off
    UseBusSignalLabels (on/off)    : off
    StoreOffsets (true/false)      : false
    StoreAdvisor (true/false)      : false

Options for 'blockbyblock' algorithm
    BlockReduction (on/off)                    : on
    IgnoreDiscreteStates (on/off)              : off
    RateConversionMethod (zoh/tustin/prewarp/  : zoh
                          upsampling_zoh/
                          upsampling_tustin/
                          upsampling_prewarp
    PreWarpFreq                                : 10
    UseExactDelayModel (on/off)                : off
    AreParamsTunable (true/false)              : true

Options for 'numericalpert' algorithm
    NumericalPertRel : 1.000000e-05
    NumericalXPert   : []
    NumericalUPert   : []
```

The following command performs a linearization using the zero-order hold method. Because the linearization includes the `Zero-Order Hold` block, the sample time of the linearization is 0.01.

```
sys_zoh = linearize(model,io,opt);
```

The following commands change the rate conversion method to the Tustin (Bilinear transformation) method and then linearize using this method. The sample time of this linearized model is also 0.01.

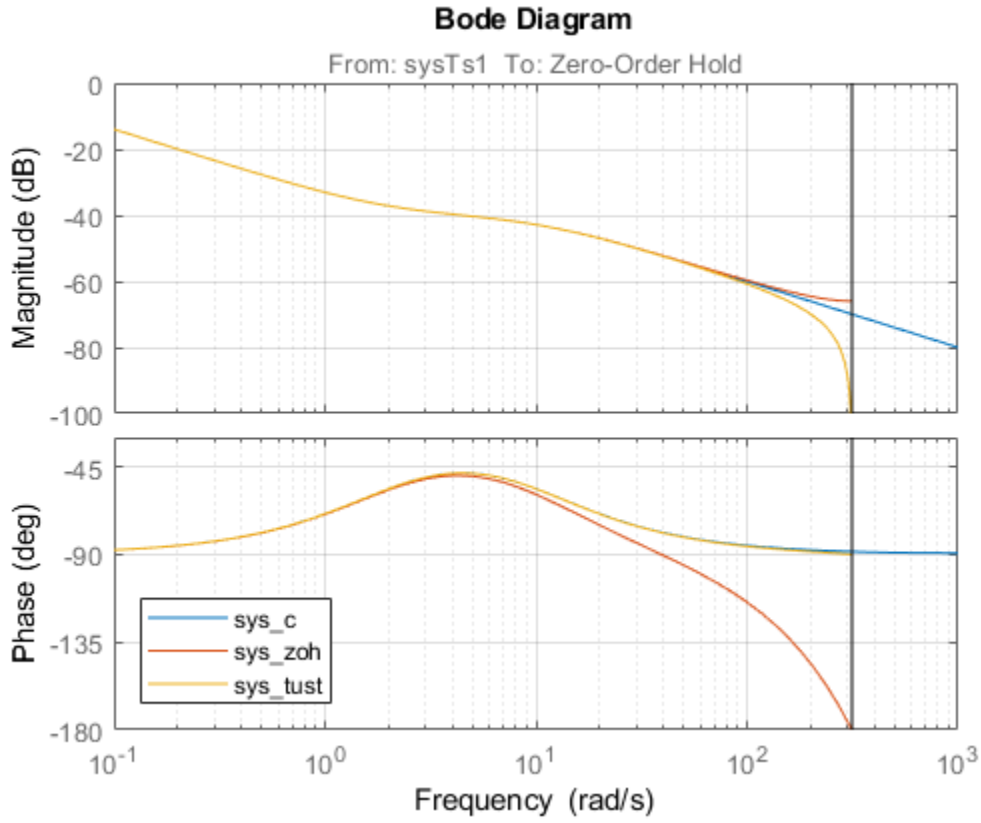
```
opt.RateConversionMethod = 'tustin';
sys_tust = linearize(model,io,opt);
```

It is also possible to create a continuous-time linearized model by specifying the sample time as 0 in the options object. The rate conversion method still creates a discrete-time linearized model but then converts the discrete-time model to a continuous-time model.

```
opt.SampleTime = 0;
sys_c = linearize(model,io,opt);
```

The Bode plots for the three linearizations show the effects of the two rate conversion methods. In this example, the Tustin rate conversion method gives the most accurate representation of the phase response of the continuous system and the zero-order hold gives the best match to the magnitude response.

```
p = bodeoptions('cstprefs');
p.YLimMode = {'manual'};
p.YLim = {[-100 0];[-180 -30]};
p.Grid = 'on';
bodeplot(sys_c,sys_zoh,sys_tust,p);
h = legend('sys_c','sys_zoh','sys_tust','Location','SouthWest');
h.Interpreter = 'none';
```



Close the models:

```
bdclose('scdmrate');
bdclose('scdmrate_ol');
```

Linearization of Multirate Models

This example shows the process that the command linearize uses when extracting a linear model of a nonlinear multirate Simulink model. To illustrate the concepts, the process is first performed using functions from the Control System Toolbox before it is repeated using the linearize command.

Example Problem

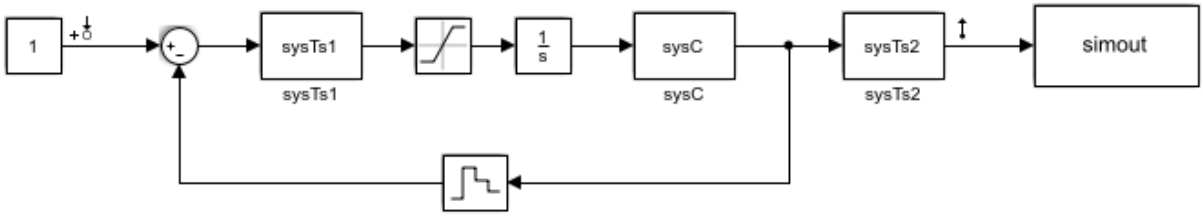
In the Simulink model `scdmrate.slx` there are three different sample rates specified in five blocks. These blocks are:

- `sysC` - a continuous linear block,
- `Integrator` - a continuous integrator,
- `sysTs1` - a block that has a sample time of 0.01 seconds,
- `sysTs2` - a block that has a sample time of 0.025 seconds, and
- `Zero-Order Hold` - a block that samples the incoming signal at 0.01 seconds.

```
sysC = zpk(-2,-10,0.1);
Integrator = zpk([],0,1);
sysTs1 = zpk(-0.7463,[0.4251 0.9735],0.2212,0.01);
sysTs2 = zpk([],0.7788,0.2212,0.025);
```

The model below shows how the blocks are connected.

```
scdmrate
```

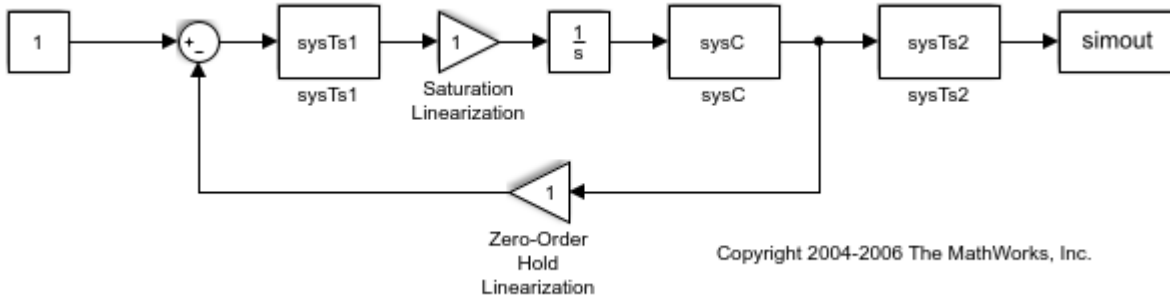


In this example we linearize the model between the output of the Constant block and the output of the block `sysTs2`.

Step 1: Linearizing the Blocks in the Model

The first step of the linearization is to linearize each block in the model. The linearization of the Saturation and Zero-Order Hold blocks is 1. The LTI blocks are already linear and therefore remain the same. The new model with linearized blocks is shown below.

```
scdmratestep1
```



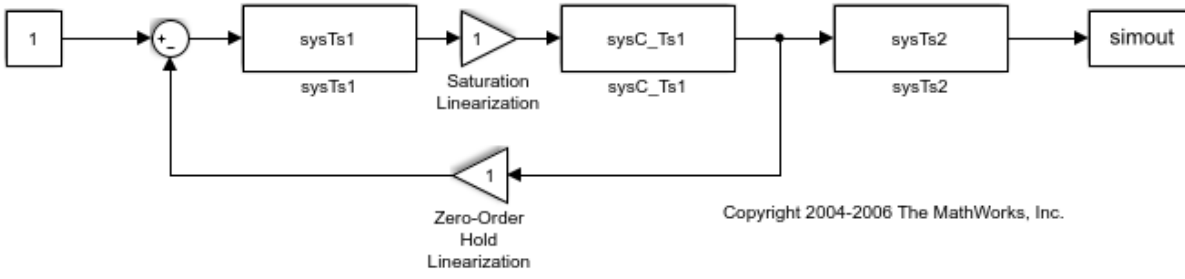
Step 2: Rate Conversions

Because the blocks in the model contain different sample rates, it is not possible to create a single-rate linearized model for the system without first using rate conversion functions to convert the various sample rates to a representative single rate. The rate conversion functions use an iterative method. The iterations begin with a least common multiple of the sample times in the model. In this example the sample times are 0, 0.01, and 0.025 seconds which yields a least common multiple of 0.05. The rate conversion functions then take the combination of blocks with the fastest sample rate and resample them at the next fastest sample rate. In this example the first iteration converts the combination of the linearized continuous time blocks, `sysC` and `integrator` to a sample time of 0.01 using a zero order hold continuous to discrete conversion.

```
sysC_Ts1 = c2d(sysC*Integrator,0.01);
```

The blocks `sysC` and `Integrator` are now replaced by `sysC_Ts1`.

```
scdmratestep2
```



The next iteration converts all the blocks with a sample time of 0.01 to a sample time of 0.025. First, the following command represents the combination of these blocks by closing the feedback loop.

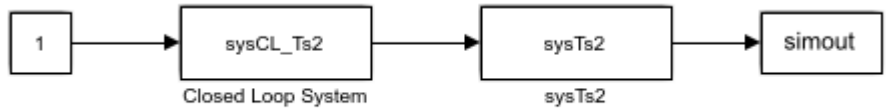
```
sysCL = feedback(sysTs1*sysC_Ts1,1);
```

Next, a zero-order hold method converts the closed loop system, sysCL, from a sample rate of 0.01 to 0.025.

```
sysCL_Ts2 = d2d(sysCL,0.025);
```

The system `sysCL_Ts2` then replaces the feedback loop in the model.

```
scdmratestep3
```



The final iteration re-samples the combination of the closed loop system and the block `sysTs2` from a rate of 0.025 seconds to a rate of 0.05 seconds.

```
sys_L = d2d(sysCL_Ts2*sysTs2,0.05)
```

```
sys_L =

   0.0001057 (z+22.76) (z+0.912) (z-0.9048) (z+0.06495)
  ------------------------------------------------------
  (z-0.01373) (z-0.6065) (z-0.6386) (z-0.8588) (z-0.9754)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
```

Linearizing the Model Using Simulink Control Design Commands

We can reproduce these results using the command line interface of Simulink Control Design.

```
model = 'scdmrate';
io(1) = linio('scdmrate/Constant',1,'input');
```

```
io(2) = linio('scdmrate/sysTs2',1,'openoutput');
sys = zpk(linearize(model,io))


sys =

  From input "Constant" to output "sysTs2":
   0.0001057 (z+22.76) (z+0.912) (z-0.9048) (z+0.06495)
  ---------------------------------------------------------
  (z-0.6065) (z-0.6386) (z-0.8588) (z-0.9754) (z-0.01373)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
```

See Also

Apps

Linear Analysis Tool

Functions

`linearize` | `linearizeOptions`

Change Perturbation Level of Blocks Perturbed During Linearization

Blocks that do not have preprogrammed analytic Jacobians linearize using numerical perturbation.

In this section...

Change Block Perturbation Level

This example shows how to change the perturbation level to the Magnetic Ball Plant block in the `magball` model. Changing the perturbation level changes the linearization results.

The default perturbation size is $10^{-5}(1+|x|)$, where x is the operating point value of the perturbed state or the input.

Open the model before changing the perturbation level.

To change the perturbation level of the states to $10^{-7}\left(1+|x|\right)$, where x is the state value, type:

```
blockname='magball/Magnetic Ball Plant'
set_param(blockname,'StatePerturbationForJacobian','1e-7')
```

To change the perturbation level of the input to $10^{-7}\left(1+|x|\right)$, where x is the input signal value:

1. Open the system and get the block port handles.

   ```
   sys = 'magball';
   open_system(sys)
   blockname = 'magball/Magnetic Ball Plant';
   ph = get_param(blockname,'PortHandles')
   ```

2. Get the handle to the inport value.

```
p_in = ph.Inport(1)
```

3 Set the inport perturbation level.

```
set_param(p_in,'PerturbationForJacobian','1e-7')
```

Perturbation Levels of Integer Valued Blocks

A custom block that requires integer input ports for indexing might have linearization issues when this block:

- Does not support small perturbations in the input value
- Accepts double-precision inputs

To fix the problem, try setting the perturbation level of such a block to zero (which sets the block linearization to a gain of 1).

Linearize Blocks with Nondouble Precision Data Type Signals

You can linearize blocks that have nondouble precision data type signals as either inputs or outputs, and have no preprogrammed exact linearization. Without additional configuration, such blocks automatically linearize to zero. For example, logical operator blocks have Boolean outputs and linearize to 0.

Linearizing blocks that have nondouble precision data type signals requires converting all signals to double precision. This approach only works when your model can run correctly in full double precision.

When you have only a few blocks impacted by the nondouble precision data types, use a Data Type Conversion block to fix this issue.

When you have many nondouble precision signals, you can override all data types with double precision using the Fixed Point Tool.

In this section...
"Overriding Data Types Using Data Type Conversion Block" on page 2-183
"Overriding Data Types Using Fixed Point Tool" on page 2-184

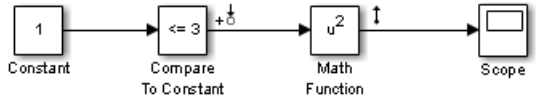
Overriding Data Types Using Data Type Conversion Block

Convert individual signals to double precision before linearizing the model by inserting a Data Type Conversion block. This approach works well for model that have only a few affected blocks.

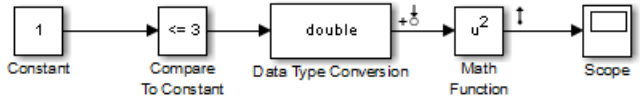
After linearizing the model, remove the Data Type Conversion block from your model.

Note Overriding nondouble data types is not appropriate when the model relies on these data types, such as relying on integer data types to perform truncation from floats.

For example, consider the model configured to linearize the Square block at an operating point where the input is 1. The resulting linearized model should be 2, but the input to the Square block is Boolean. This signal of nondouble precision date type results in linearization of zero.



In this case, inserting a Data Type Conversion block converts the input signal to the Square block to double precision.



Overriding Data Types Using Fixed Point Tool

When you linearize a model that contains nondouble data types but still runs correctly in full double precision, you can override all data types with doubles using the Fixed Point Tool. Use this approach when you have many nondouble precision signals.

After linearizing the model, restore your original settings.

Note Overriding nondouble data types is not appropriate when the model relies on these data types, such as relying on integer data types to perform truncation from floats.

1. In the Simulink model, select **Analysis** > **Fixed Point Tool**.

   The Fixed Point Tool opens.
2. In the **Data type override** menu, select **Double**.

   This setting uses double precision values for all signals during linearization.
3. Restore settings when linearization completes.

Linearize Event-Based Subsystems (Externally Scheduled Subsystems)

In this section...

Linearizing Event-Based Subsystems

Event-based subsystems (triggered subsystems) and other event-based models require special handling during linearization.

Executing a triggered subsystem depends on previous signal events, such as zero crossings. However, because linearization occurs at a specific moment in time, the trigger event never happens.

An example of an event-based subsystem is an internal combustion (IC) engine. When an engine piston approaches the top of a compression stroke, a spark causes combustion. The timing of the spark for combustion is dependent on the speed and the position of the engine crankshaft.

In the `scdspeed` model, triggered subsystems generate events when the pistons reach both the top and bottom of the compression stroke. Linearization in the presence of such triggered subsystems is not meaningful.

Approaches for Linearizing Event-Based Subsystems

You can obtain a meaningful linearization of triggered subsystems, while still preserving the simulation behavior, by recasting the event-based dynamics as one of the following:

- Lumped average model that approximates the event-based behavior over time.
- Periodic function call subsystem, with Normal simulation mode.

In the case of periodical function call subsystems, the subsystem linearizes to the sampling at which the subsystem is periodically executed.

In many control applications, the controller is implemented as a discrete controller, but the execution of the controller is driven by an external scheduler. You can use such linearized plant models when the controller subsystem is marked as a Periodic Function call subsystem.

If recasting event-based dynamics does not produce good linearization results, try frequency response estimation. See “Estimate Frequency Response Using Linear Analysis Tool” on page 5-26.

Periodic Function Call Subsystems for Modeling Event-Based Subsystems

This example shows how to use periodic function call subsystems to approximate event-based dynamics for linearization.

1 Open Simulink model.

```
sys = 'scdPeriodicFcnCall';
open_system(sys)
```

2 Linearize model at the model operating point.

```
io = getlinio(sys);
linsys = linearize(sys,io)
```

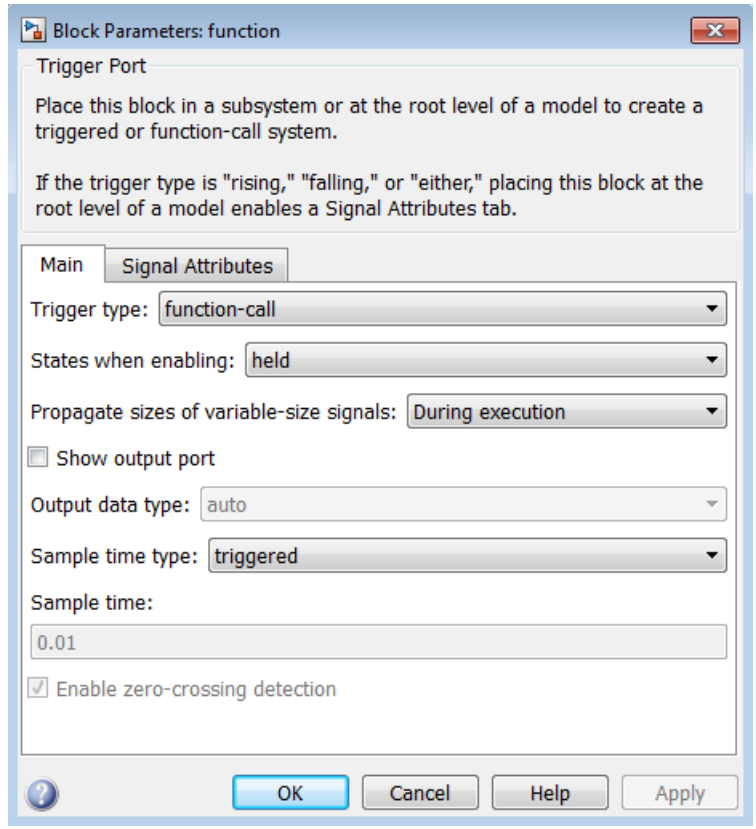
The linearization is zero because the subsystem is not a periodic function call.

```
D =
                  Desired  Wat
   Water-Tank S             0
 Static gain.
```

Now, specify the Externally Scheduled Controller block as a Periodic Function Call Subsystem.

3 Double-click the Externally Scheduled Controller (Function-Call Subsystem) block.

Double-click the function block to open the Block Parameters dialog box.



4 Set **Sample time type** to be `periodic`.

Leave the **Sample time** value as 0.01, which represents the sample time of the function call.

5 Linearize the model.

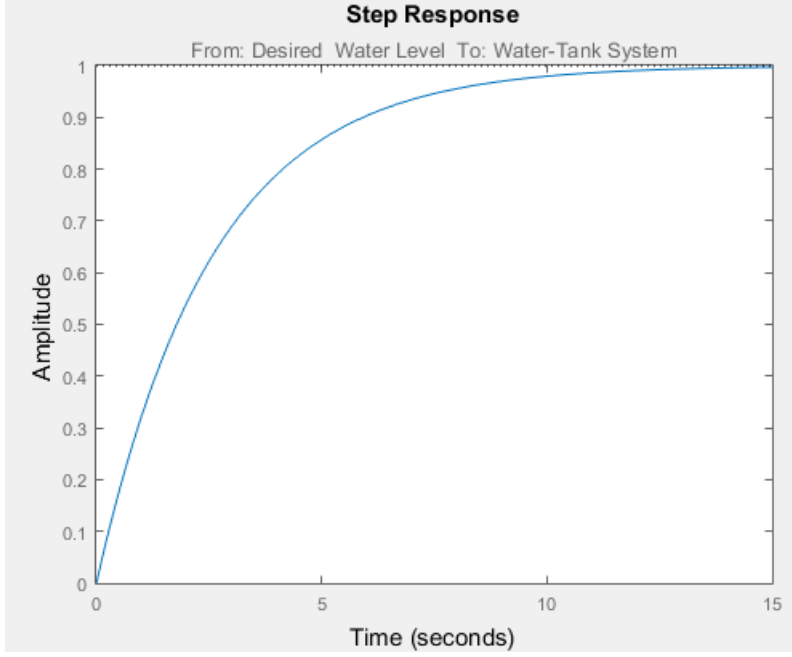
```
linsys2 = linearize(sys,io)
```

```
A =
                        H  Integrator
   H                0.9956    0.002499
   Integrator   -0.0007774           1

B =
                Desired  Wat
```

```
   H               0.003886
   Integrator     0.0007774

C =
                           H  Integrator
   Water-Tank S            1           0

D =
                  Desired  Wat
   Water-Tank S             0

Sampling time: 0.01
Discrete-time model.
```

6 Plot step response.

```
step(linsys2)
```



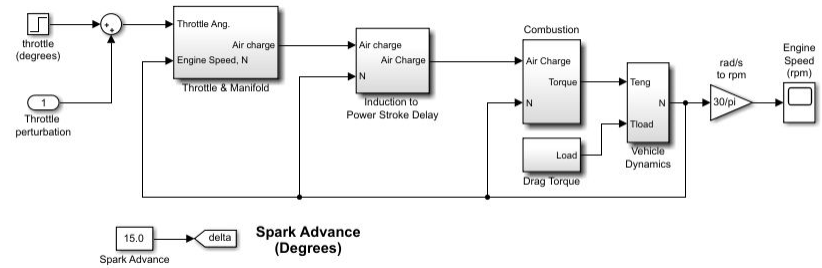
7 Close the model.

```
bdclose(sys);
```

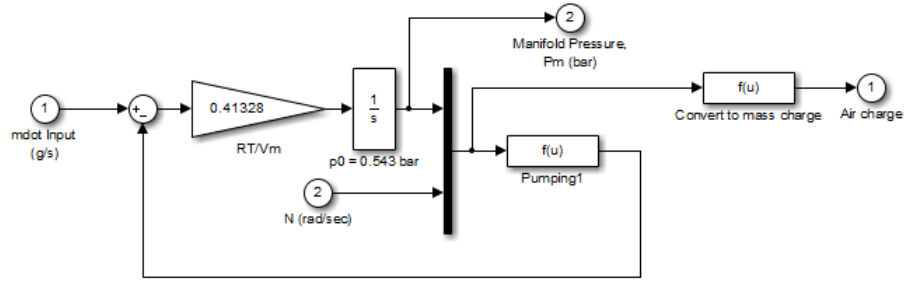
Approximating Event-Based Subsystems Using Curve Fitting (Lump-Average Model)

This example shows how to use curve fitting to approximate event-based dynamics of an engine.

The `scdspeed` model linearizes to zero because `scdspeed/Throttle & Manifold/Intake Manifold` is an event-triggered subsystem.



You can approximate the event-based dynamics of the `scdspeed/Throttle & Manifold/Intake Manifold` subsystem by adding the Convert to mass charge block inside the subsystem.



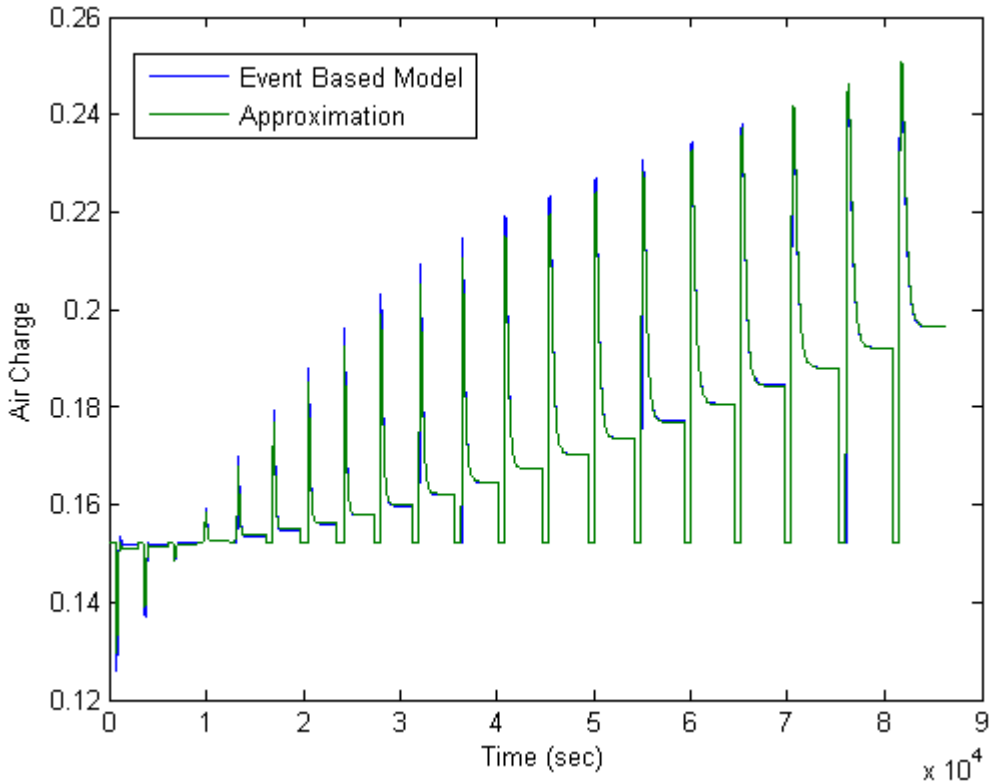
The Convert to mass charge block approximates the relationship between Air Charge, Manifold Pressure, and Engine Speed as a quadratic polynomial.

$$Air\,Charge = p_1 \times Engine\,Speed + p_2 \times Manifold\ Pressure + p_3 \times (Manifold\ Pressure)^2 + p_4 \times Manifold\ Pressure \times Engine\ Speed + p_5$$

If measured data for internal signals is not available, use simulation data from the original model to compute the unknown parameters p_1, p_2, p_3, p_4, and p_5 using a least squares fitting technique.

When you have measured data for internal signals, you can use the Simulink Design Optimization™ software to compute the unknown parameters. See "Engine Speed Model Parameter Estimation" (Simulink Design Optimization) to learn more about computing model parameters, linearizing this approximated model, and designing a feedback controlled for the linear model.

The next figure compares the simulations of the original event-based model and the approximated model. Each of the pulses corresponds to a step change in the engine speed. The size of the step change is between 1500 and 5500. Thus, you can use the approximated model to accurately simulate and linearize the engine between 1500 RPM and 5500 RPM.



See Also

More About

- “Linearization of Models with Model References”

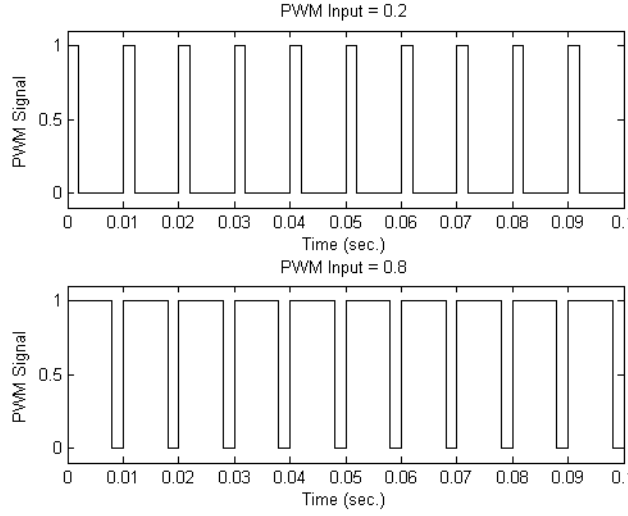
Configure Models with Pulse Width Modulation (PWM) Signals

This example shows how to configure models that use Pulse Width Modulation (PWM) input signals for linearization. For linearization, specify a custom linearization of the subsystem that takes the DC signal to be a gain of 1.

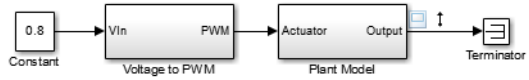
Many industrial applications use Pulse Width Modulation (PWM) signals because such signals are robust in the presence of noise.

The next figure shows two PWM signals. In the top plot, a PWM signal with a 20% duty cycle represents a 0.2 V DC signal. A 20% duty cycle corresponding to 1 V signal for 20% of the cycle, followed by a value of 0 V signal for 80% of the cycle. The average signal value is 0.2 V.

In the bottom plot, a PWM signal with an 80% duty cycle represent a 0.8 V DC signal.

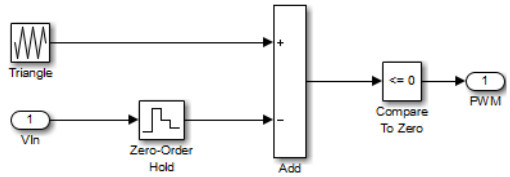


For example, in the `scdpwm` model, a PWM signal is converted to a constant signal.



When linearizing a model containing PWM signals there are two effects of linearization you should consider:

- The signal level at the operating point is one of the discrete values within the PWM signal, not the DC signal value. For example, in the model above, the signal level is either 0 or 1, not 0.8. This change in operating point affects the linearized model.
- The creation of the PWM signal within the subsystem `Voltage to PWM`, shown in the next figure, uses the Compare to Zero block. Such comparator blocks do not linearize well due to their discontinuities and the nondouble outputs.



See Also

More About

- "Specifying Custom Linearizations for Simulink Blocks"
- "Specifying Linearization for Model Components Using System Identification" on page 2-199

Linearize Simscape Networks

You can linearize models with Simscape components using Simulink Control Design software. Typically, some states in a Simscape network have dependencies on other states through constraints.

Find Steady-State Operating Point

To find a steady-state operating point at which to linearize a Simscape model, you can use:

- Optimization-based trimming — Specify constraints on model inputs, outputs, or states, and compute a steady-state operating point that satisfies these constraints.

  To produce better trimming results for Simscape models, you can use projection-based trim optimizers. These optimizers enforce the consistency of the model initial condition at each evaluation of the objective function or nonlinear constraint function.
- Simulation snapshots — Specify model initial conditions near an expected equilibrium point, and simulate the model until it reaches steady state.

For more information, see “Find Steady-State Operating Points for Simscape Models” on page 1-95.

Specify Analysis Points

To linearize your model, you must specify the portion of the model you want to linearize using linear analysis points; that is, linearization inputs and outputs, and loop openings. You can only add analysis points to Simulink signals.

To add a linearization input or loop opening to the input of a Simscape component, first convert the Simulink signal using a Simulink-PS Converter block.

To add a linearization output or loop opening to the output of a Simscape component, first convert the Simscape signal using a PS-Simulink Converter block.

For more information on adding linear analysis points, see “Specify Portion of Model to Linearize” on page 2-13.

Linearize Model

After you specify a steady-state operating point and linear analysis points, you can linearize your Simscape model using:

- The **Linear Analysis Tool**.
- The `linearize` function.
- An `slLinearizer` interface.

For general linearization examples, see “Linearize Simulink Model at Model Operating Point” on page 2-69 and “Linearize at Trimmed Operating Point” on page 2-85.

Troubleshoot Simscape Network Linearizations

Simscape networks can commonly linearize to zero when a set of the system equation Jacobians are zero at a given operating condition. Usually, poor initial conditions of the network states cause these zero linearizations.

Zero Linearization Example

Consider a system where the mass flow rate from a variable orifice is controlling the position of a piston. The mass flow rate equation of the variable orifice is:

$$q = C_d A \sqrt{\frac{2}{\mu}} \left(\frac{p}{p^2 + p_{cr}^2} \right)^{0.25}$$

Where:

- q is the mass flow rate.
- C_d is the discharge coefficient.
- A is the area of the variable orifice opening.
- μ is the fluid density.
- p is the pressure drop across the orifice, $p = p_a - p_b$.
- p_{cr} is the critical pressure, which is a function of p_a and p_b.

The control variable for this system is the orifice area, A, which controls the mass flow rate. The Jacobian of the mass flow rate with respect to the control variable is:

$$\frac{\partial q}{\partial A} = C_d \sqrt{\frac{2}{\mu}} \left(\frac{p}{p^2 + p_{cr}^2} \right)^{0.25}$$

The linearized mass flow rate equation is:

$$\bar{q} = C_d \sqrt{\frac{2}{\mu}} \left(\frac{p}{p^2 + p_{cr}^2} \right)^{0.25} \bar{A} + \frac{\partial q}{\partial \mu} \bar{\mu}$$
$$+ \left(\frac{\partial q}{\partial p_{cr}} \frac{\partial p_{cr}}{\partial p_a} + \frac{\partial q}{\partial p} \right) \bar{p}_a + \left(\frac{\partial q}{\partial p_{cr}} \frac{\partial p_{cr}}{\partial p_b} - \frac{\partial q}{\partial p} \right) \bar{p}_b$$

where $\bar{\bullet}$ represents a deviation from the nominal variable.

In the linearized equation, if the nominal pressure drop p across the orifice is zero, then $\bar{A}$ has no influence on $\bar{q}$. That is, if the instantaneous pressure drop across the orifice is zero, the orifice area has no influence on the mass flow rate. Therefore, you cannot control the piston position using the orifice area control variable.

To avoid this condition, linearize the model about an operating point where the pressure drop over the orifice is nonzero ($p_a \neq p_b$).

Troubleshooting Tips

To fix linearization problems caused by poor initial conditions of network states, you can:

1. Linearize the system at a snapshot operating point or trimmed operating point. When possible, this approach is recommended.
2. Find and modify the problematic states of the operating point. This option can be difficult for models with many states.

Using the first approach, you can ensure that the model states are consistent via the Simulink and Simscape simulation engine. Simscape initial conditions are not necessarily in a consistent state. The Simscape engine places them in a consistent state during simulation and for trimming using the Simscape trim solvers.

A common workflow is to simulate your model, observe at what time the model satisfies the operating condition at which you want to linearize, then take a simulation snapshot. Alternatively, you can trim the model about the condition you are interested in. In either case, the network states are in a consistent condition, which solves most poor linearization issues.

Using the second approach, you search through the physical network states to find conditions that can create a zero Jacobian. This approach can require some intuition about the dynamics of the physical components in your model. As a starting point, search for states that are zero and that interact directly with nonlinear physical elements, such as the variable orifice in the preceding example.

To search the physical states, you can use the Linearization Advisor, which collects diagnostic information during linearization. The Linearization Advisor does not provide diagnostic information on a component-level basis. Instead, it groups diagnostic information for multiple Simscape components together.

1. Linearize your model with the Linearization Advisor enabled, and extract the `LinearizationAdvisor` object.

   ```
   opt = linearizeOptions('StoreAdvisor',true);
   [linsys,linop,info] = linearize(mdl,io,op,opt);
   advisor = info.Advisor;
   ```

2. Create a custom query object, and search the diagnostic information for Simscape blocks.

   ```
   qSS = linqueryIsBlockType('Simscape');
   advSS = find(advisor,qSS);
   ```

3. Obtain `BlockDiagnostic` objects for all the Simscape blocks associated with model states. The block paths for these blocks contain the text `EVAL_KEY/STATE`.

   ```
   paths = getBlockPaths(advisor);
   index = contains(paths,'EVAL_KEY/STATE');
   diag = getBlockInfo(advSS,index);
   ```

4. To find problematic state values, check the block operating point in each `BlockDiagnostic` object.

   ```
   diag(i).OperatingPoint
   ```

Once you find a problematic state, you can change the value of the state in the model operating point, or create an operating point using `operpoint`.

You can also search the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Apps

Linear Analysis Tool

Functions

`linearize` | `slLinearizer`

More About

- “Specify Portion of Model to Linearize” on page 2-13
- “Find Steady-State Operating Points for Simscape Models” on page 1-95
- “Linearize Simulink Model at Model Operating Point” on page 2-69

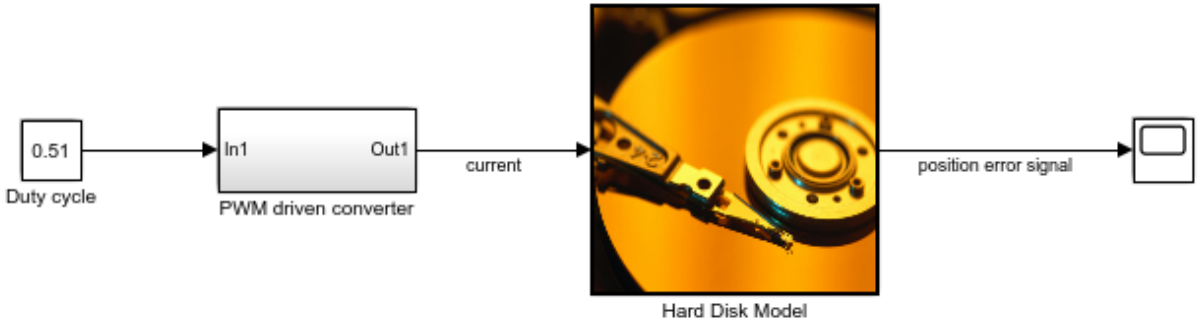
Specifying Linearization for Model Components Using System Identification

This example shows how to use System Identification Toolbox to identify a linear system for a model component that does not linearize well and use the identified system to specify its linearization. Note that running this example requires Simscape Power Systems in addition to System Identification Toolbox.

Linearizing Hard Drive Model

Open the simulink model for the hard drive.

```
model = 'scdpwmharddrive';
open_system(model);
```



In this model, the hard drive plant is driven by a current source. The current source is implemented by a circuit that is driven by a Pulse Width Modulation (PWM) signal so that its output can be adjusted by the duty cycle. For details of the hard drive model, see the example "Digital Servo Control of a Hard-Disk Drive" in Control System Toolbox™ examples.

PWM-driven circuits usually have high frequency switching components, such as the MOSFET transistor in this model, whose average behavior is not well defined. Thus, exact linearization of this type of circuits is problematic. When you linearize the model from duty cycle input to the position error, the result is zero.

```
io(1) = linio('scdpwmharddrive/Duty cycle',1,'input');
io(2) = linio('scdpwmharddrive/Hard Disk Model',1,'output');
sys = linearize(model,io)
```

```
sys =

  D =
                 Duty cycle
   position err           0

Static gain.
```

Finding a Linear Model for PWM Component

You can use frequency response estimation to obtain the frequency response of the PWM-driven current source and use the result to identify a linear model for it. The current signal has a discrete sample time of 1e-7. Thus, you need to use a fixed sample time sinestream signal. Create a signal that has frequencies between 2K and 200K rad/s.

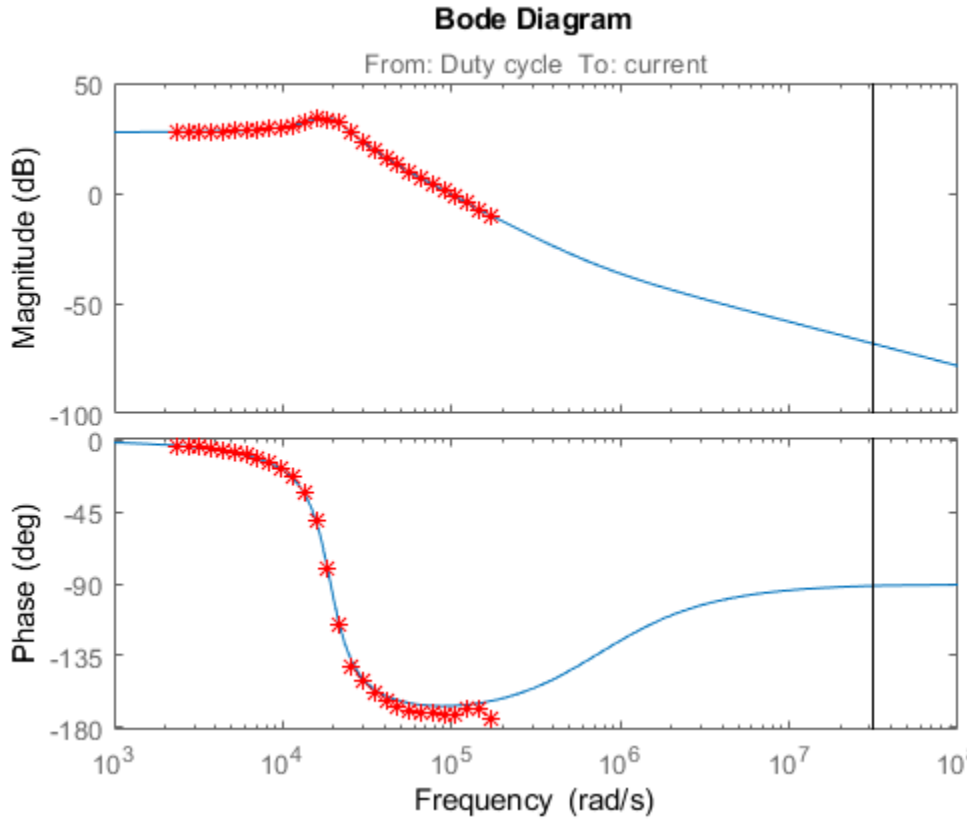
```
idinput = frest.createFixedTsSinestream(Ts,{2000,200000});
idinput.Amplitude = 0.1;
```

You can then define the input and output points for PWM-driven circuit and run the frequency response estimation with the sinestream signal.

```
pwm_io(1) = linio('scdpwmharddrive/Duty cycle',1,'input');
pwm_io(2) = linio('scdpwmharddrive/PWM driven converter',1,'openoutput');
sysfrd = frestimate(model,pwm_io,idinput);
```

Using the `N4SID` command from System Identification Toolbox, you can identify a second-order model using the frequency response data. Then, compare the identified model to the original frequency response data.

```
sysid = ss(tfest(idfrd(sysfrd),2));
bode(sysid,sysfrd,'r*');
```



We used frequency response data with frequencies between 2K and 200K rad/s. The identified model has a flat magnitude response for frequencies smaller than 2K. However, our estimation did not have include for those frequencies. Assume that you would like to make sure the response is flat by checking the frequency response for 20 and 200 rad/s. To do so, create another input signal with those frequencies in it.

```
lowfreq = [20 200];
inputlow = frest.createFixedTsSinestream(Ts,lowfreq)
```

```
The sinestream input signal:

      Frequency           : [20 200] (rad/s)
      Amplitude           : 1e-05
```

```
SamplesPerPeriod    : [3141593 314159]
NumPeriods          : 4
RampPeriods         : 0
FreqUnits (rad/s,Hz): rad/s
SettlingPeriods     : 1
ApplyFilteringInFRESTIMATE (on/off)    : on
SimulationOrder (Sequential/OneAtATime): Sequential
```

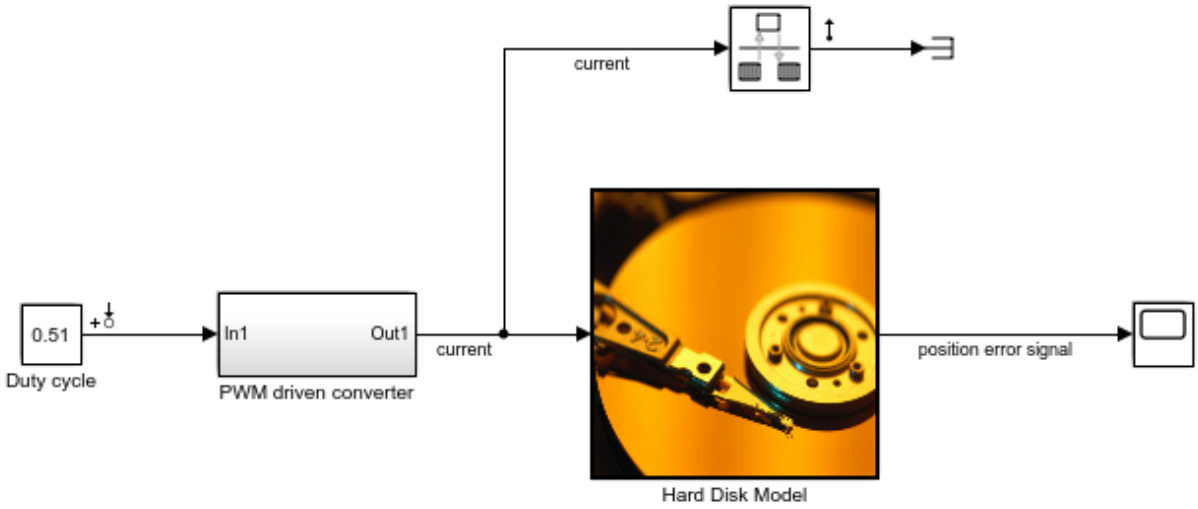
In the input signal parameters, we can see that having a very fast sample rate of 1e-7 seconds (10 MHz sampling frequency) for the frequencies 20 and 200 rad/s cause high `SamplesPerPeriod` values of 3141593 and 314159. Considering that each frequency has 4 periods, frequency response estimation would log output data with around 14 millions samples. This would require a lot of memory and it is quite likely that you might run into memory issues running the estimation.

Obviously, you do not need such a high sampling rate for analyzing 20 and 200 rad/s frequencies. You can use a smaller sampling rate to avoid memory issues:

```
Tslow = 1e-4;
wslow = 2*pi/Tslow;
inputlow = frest.createFixedTsSinestream(Tslow,wslow./round(wslow./lowfreq));
inputlow.Amplitude = 0.1;
```

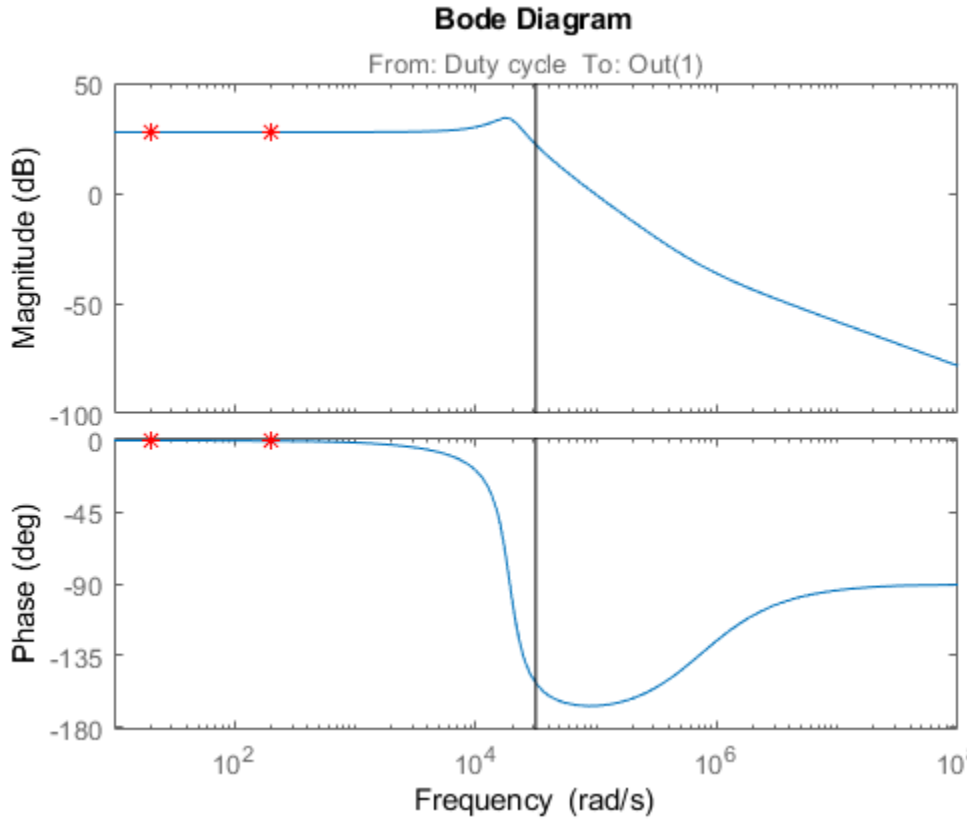
To make the model compatible with the smaller sampling rate, resample the output data point using a rate transition block as in the modified model:

```
modellow = 'scdpwmharddrive_lowfreq';
open_system(modellow);
```



You can now run the analysis for the low frequencies and compare it against identification result.

```
load scdpwmharddrive_lowfreqresults.mat
% sysfrdlow = frestimate(modellow,getlinio(modellow),inputlow);
bode(sysid,sysfrdlow,'r*');
bdclose(modellow);
```



Specifying the Linearization for PWM Component

As you verified using the frequency response estimation, the low-frequency dynamics of the PWM-driven component are captured well by the identified system. Now you can make linearization use this system as the linearization of the PWM-driven component. To do so, specify block linearization of that subsystem as follows:

```
pwmblock = 'scdpwmharddrive/PWM driven converter';
set_param(pwmblock,'SCDEnableBlockLinearizationSpecification','on');
rep = struct('Specification','sysid',...
             'Type','Expression',...
             'ParameterNames','',...
             'ParameterValues','');
```

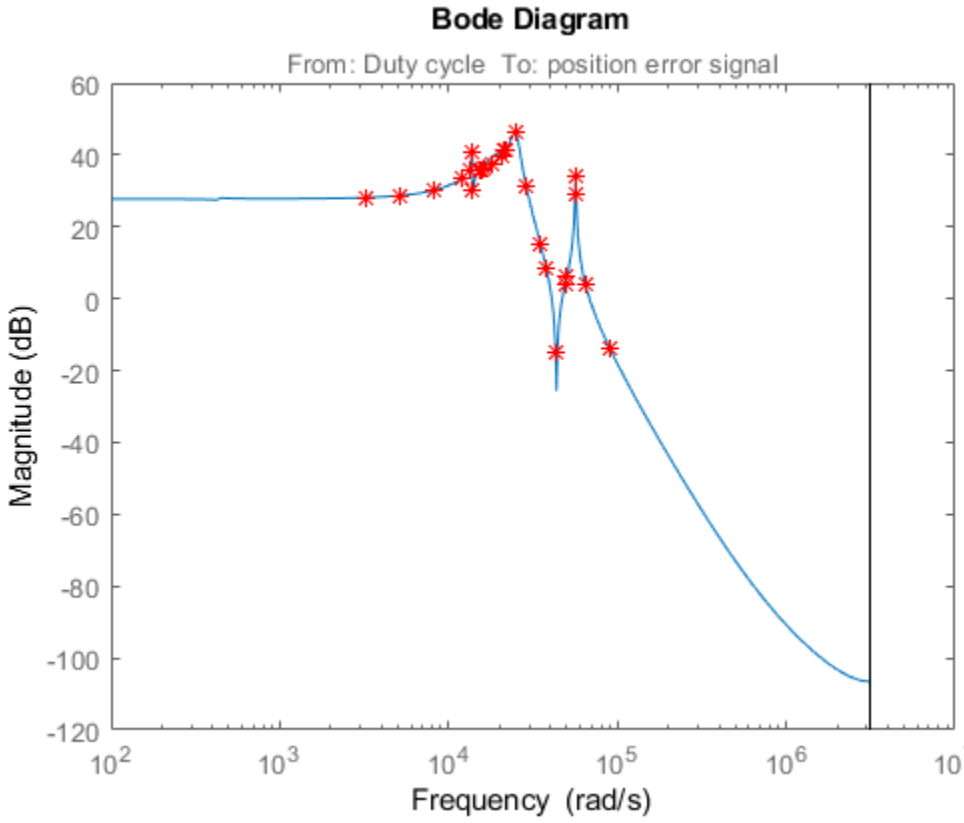
```
set_param(pwmblock,'SCDBlockLinearizationSpecification',rep);
set_param('scdpwmharddrive/Duty cycle','SampleTime','Ts_plant');
```

Linearizing the model after specifying the linearization of the PWM component gives us a non-zero result:

```
sys = linearize(model,io);
```

You might still like to validate this linearization result using frequency response estimation. Doing this as below verifies that our linearization result is quite accurate and all the resonances exist in the actual dynamics of the model.

```
valinput = frest.Sinestream(sys);
valinput = fselect(valinput,3e3,1e5);
valinput.Amplitude = 0.1;
sysval = frestimate(model,io,valinput);
bodemag(sys,sysval,'r*');
```



Close the model:

```
bdclose('scdpwmharddrive');
```

See Also

`frestimate` | `tfest`

More About

- “Configure Models with Pulse Width Modulation (PWM) Signals” on page 2-192

Exact Linearization Algorithm

In this section...

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations.

The software determines the input and state levels for each block from the operating point, and requests the Jacobian for these levels from each block.

For some blocks, the software cannot compute an analytical linearization. For example:

- Some nonlinearities do not have a defined Jacobian.
- Some discrete blocks, such as state charts and triggered subsystems, tend to linearize to zero.
- Some blocks do not implement a Jacobian.
- Custom blocks, such as S-Function blocks and MATLAB Function blocks, do not have analytical Jacobians.

You can specify a custom linearization for any such blocks for which you know the expected linearization. If you do not specify a custom linearization, the software finds the linearization by perturbing the block inputs and states and measuring the response to these perturbations. For more information, see “Perturbation of Individual Blocks” on page 2-209.

Continuous-Time Models

Simulink Control Design software lets you linearize continuous-time nonlinear systems. The resulting linearized model is in state-space form.

In continuous time, the state space equations of a nonlinear system are:

$$\dot{x}(t) = f\left(x(t), u(t), t\right)$$
$$y(t) = g\left(x(t), u(t), t\right)$$

where $x(t)$ are the system states, $u(t)$ are the input signals, and $y(t)$ are the output signals.

To describe the linearized model, define a new set of variables of the states, inputs, and outputs centered about the operating point:

$$\delta x(t) = x(t) - x_0$$
$$\delta u(t) = u(t) - u_0$$
$$\delta y(t) = y(t) - y_0$$

The output of the system at the operating point is $y(t_0)=g(x_0,u_0,t_0)=y_0$.

The linearized state-space equations in terms of $\delta x(t)$, $\delta u(t)$, and $\delta y(t)$ are:

$$\delta \dot{x}(t) = A\delta x(t) + B\delta u(t)$$
$$\delta y(t) = C\delta x(t) + D\delta u(t)$$

where A, B, C, and D are constant coefficient matrices. These matrices are the Jacobians of the system, evaluated at the operating point:

$$A = \left.\frac{\partial f}{\partial x}\right|_{t_0, x_0, u_0} \qquad B = \left.\frac{\partial f}{\partial u}\right|_{t_0, x_0, u_0}$$
$$C = \left.\frac{\partial g}{\partial x}\right|_{t_0, x_0, u_0} \qquad D = \left.\frac{\partial g}{\partial u}\right|_{t_0, x_0, u_0}$$

This linear time-invariant approximation to the nonlinear system is valid in a region around the operating point at $t=t_0$, $x(t_0)=x_0$, and $u(t_0)=u_0$. In other words, if the values of the system states, $x(t)$, and inputs, $u(t)$, are close enough to the operating point, the system behaves approximately linearly.

The transfer function of the linearized model is the ratio of the Laplace transform of $\delta y(t)$ and the Laplace transform of $\delta u(t)$:

$$P_{lin}(s) = \frac{\delta Y(s)}{\delta U(s)}$$

Multirate Models

Simulink Control Design software lets you linearize multirate nonlinear systems. The resulting linearized model is in state-space form.

Multirate models include states with different sampling rates. In multirate models, the state variables change values at different times and with different frequencies. Some of the variables might change continuously.

The general state-space equations of a nonlinear, multirate system are:

$$\begin{aligned}
\dot{x}(t) &= f\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
x_1(k_1+1) &= f_1\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
&\vdots \qquad\qquad\qquad\qquad \vdots \\
x_m(k_m+1) &= f_i\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
y(t) &= g\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right)
\end{aligned}$$

where k_1,..., k_m are integer values and t_{k_1} ,..., t_{k_m} are discrete times.

The linearized equations that approximate this nonlinear system as a single-rate discrete model are:

$$\begin{aligned}
\delta x_{k+1} &\approx A\delta x_k + B\delta u_k \\
\delta y_k &\approx C\delta x_k + D\delta u_k
\end{aligned}$$

The rate of the linearized model is typically the least common multiple of the sample times, which is usually the slowest sample time.

For more information, see “Linearization of Multirate Models”.

Perturbation of Individual Blocks

Simulink Control Design software linearizes blocks that do not have a preprogrammed linearization using numerical perturbation. The software computes the block linearization by numerically perturbing the states and inputs of the block about the operating point of the block.

The block perturbation algorithm introduces a small *perturbation* to the nonlinear block and measures the response to this perturbation. The default difference between the

perturbed value and the operating point value is $10^{-5}\left(1+|x|\right)$, where x is the operating point value. The software uses this perturbation and the resulting response to compute the linear state-space of this block. For information on how to change perturbation levels for individual blocks, see "Change Perturbation Level of Blocks Perturbed During Linearization" on page 2-181.

In general, a continuous-time nonlinear Simulink block in state-space form is given by:

$$\dot{x}(t) = f\left(x(t),u(t),t\right)$$
$$y(t) = g\left(x(t),u(t),t\right).$$

In these equations, $x(t)$ represents the states of the block, $u(t)$ represents the inputs of the block, and $y(t)$ represents the outputs of the block.

A linearized model of this system is valid in a small region around the operating point $t=t_0$, $x(t_0)=x_0$, $u(t_0)=u_0$, and $y(t_0)=g(x_0,u_0,t_0)=y_0$.

To describe the linearized block, define a new set of variables of the states, inputs, and outputs centered about the operating point:

$$\delta x(t) = x(t) - x_0$$
$$\delta u(t) = u(t) - u_0$$
$$\delta y(t) = y(t) - y_0$$

The linearized state-space equations in terms of these new variables are:

$$\delta\dot{x}(t) = A\delta x(t) + B\delta u(t)$$
$$\delta y(t) = C\delta x(t) + D\delta u(t)$$

A linear time-invariant approximation to the nonlinear system is valid in a region around the operating point.

The state-space matrices A, B, C, and D of this linearized model represent the Jacobians of the block.

To compute the state-space matrices during linearization, the software performs these operations:

1 Perturbs the states and inputs, one at a time, and measures the response of the system to this perturbation by computing $\delta\dot{x}$ and δy.

2 Computes the state-space matrices using the perturbation and the response.

$$A(:,i) = \frac{\dot{x}\big|_{x_{p,i}} - \dot{x}_o}{x_{p,i} - x_o}, \qquad B(:,i) = \frac{\dot{x}\big|_{u_{p,i}} - \dot{x}_o}{u_{p,i} - u_o}$$

$$C(:,i) = \frac{y\big|_{x_{p,i}} - y_o}{x_{p,i} - x_o}, \qquad D(:,i) = \frac{y\big|_{u_{p,i}} - y_o}{u_{p,i} - u_o}$$

where

- $x_{p,i}$ is the state vector whose ith component is perturbed from the operating point value.
- x_o is the state vector at the operating point.
- $u_{p,i}$ is the input vector whose ith component is perturbed from the operating point value.
- u_o is the input vector at the operating point.
- $\dot{x}\big|_{x_{p,i}}$ is the value of $\dot{x}$ at $x_{p,i}$, u_o.
- $\dot{x}\big|_{u_{p,i}}$ is the value of $\dot{x}$ at $u_{p,i}$, x_o.
- $\dot{x}_o$ is the value of $\dot{x}$ at the operating point.
- $y\big|_{x_{p,i}}$ is the value of y at $x_{p,i}$, u_o.
- $y\big|_{u_{p,i}}$ is the value of y at $u_{p,i}$, x_o.
- y_o is the value of y at the operating point.

User-Defined Blocks

All user defined blocks such as S-Function and MATLAB Function blocks, are compatible with linearization. These blocks are linearized using numerical perturbation.

User-defined blocks do not linearize when these blocks use nondouble precision data types.

See “Linearize Blocks with Nondouble Precision Data Type Signals” on page 2-183.

Look Up Tables

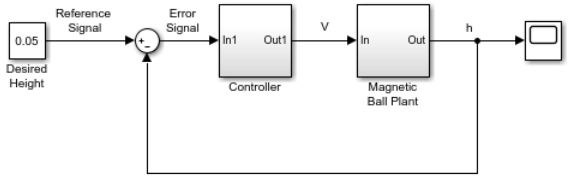
Regular look up tables are numerically perturbed. Pre-lookup tables have a preprogrammed (exact) block-by-block linearization.

Batch Linearization

What Is Batch Linearization?

Batch linearization refers to extracting multiple linearizations from a model for various combinations of I/Os, operating points, and parameter values. Batch linearization lets you analyze the time-domain, frequency-domain, and stability characteristics of your Simulink model, or portions of your model, under varying operating conditions and parameter ranges. You can use the results of batch linearization to design controllers that are robust against parameter variations, or to design gain-scheduled controllers for different operating conditions. You can also use batch linearization results to implement linear parameter varying (LPV) approximations of nonlinear systems using the LPV System block of Control System Toolbox.

To understand different types of batch linearization, consider the magnetic ball levitation model, `magball`. For more information about this model, see “magball Simulink Model”.



You can batch linearize this model by varying any combination of the following:

- I/O sets — Linearize a model using different I/Os to obtain any closed-loop or open-loop transfer function.

  For the `magball` model, some of the transfer functions that you can extract by specifying different I/O sets include:

  - Magnetic ball plant model, controller model
  - Closed-loop transfer function from the `Reference Signal` to the plant output, `h`
  - Open-loop transfer function for the controller and magnetic ball plant combined; that is, the transfer function from the `Error Signal` to `h` with the feedback loop opened
  - Output disturbance rejection model or sensitivity transfer function, obtained at the outport of `Magnetic Ball Plant` block
- Operating points — In nonlinear models, the model dynamics vary depending on the operating conditions. You can linearize a nonlinear model at different operating

points to study how model dynamics vary or to design controllers for different operating conditions.

For an example of model dynamics that vary depending on the operating point, consider a simple unforced hanging pendulum with angular position and velocity as states. This model has two equilibrium points, one when the pendulum hangs downward, which is stable, and another when the pendulum points upward, which is unstable. Linearizing close to the stable operating point produces a stable model, whereas linearizing this model close to the unstable operating point produces an unstable model.

For the `magball` model, which uses the ball height as a state, you can obtain multiple linearizations for varying initial ball heights.

- Parameters — Parameters configure a Simulink model in several ways. For example, you can use parameters to specify model coefficients or controller sample times. You can also use a discrete parameter, such as the control input to a Multiport Switch block, to control the data path within a model. Therefore, varying a parameter can serve a range of purposes, depending on how the parameter contributes to the model.

  For the `magball` model, you can vary the parameters of the PID Controller block, `Controller/PID Controller`. The linearizations obtained by varying these parameters show how the controller affects the control-system dynamics. Alternatively, you can vary the magnetic ball plant parameter values to determine the controller robustness to variations in the plant model. You can also vary the parameters of the input block, `Desired Height`, and study the effects of varying input levels on the model response.

  If the parameters affect the model operating point, you can batch trim the model using the parameter samples and then batch linearize the model at the resulting operating points.

See Also

LPV System

More About

- “Choose Batch Linearization Methods” on page 3-5
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10

Choose Batch Linearization Methods

Simulink Control Design software provides multiple tools and methods for batch linearization. Which tool and method you choose depends on your application requirements and software preferences. The following table describes the batch linearization workflows supported by Simulink Control Design software.

Application Description	Operating Point Computation Options	Linearization Workflow
Your model has more than one operating condition that does not depend on any varying model parameters. Use this approach when the model operating conditions depend only on the model states and inputs.	• Batch trim your model for multiple operating point specifications, using a single model compilation when possible. Batch trimming is not supported in the Linear Analysis Tool. • Trim the model separately for each operating point specification, which requires multiple model compilations. Use this option with the Linear Analysis Tool. • Compute operating points at multiple simulation snapshot times.	1 Compute operating points. 2 Batch linearize the model at all operating points. For an example, see: • "Batch Linearize Model at Multiple Operating Points Using linearize Command" on page 3-28

Application Description	Operating Point Computation Options	Linearization Workflow
Your model has a single operating condition, and you want to linearize the model at this operating point for varying model parameters. Examples of such an application include: • Studying the effect of component tolerances on model dynamics. • Examining controller robustness to variations in plant parameters.	• Trim the model for a single operating point specification. • Compute an operating point at a simulation snapshot time.	1 Compute operating point. 2 Define parameter values for linearization. 3 Batch linearize the model at the computed operating point for the specified parameter variations. For an example, see: • "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75 • "Batch Linearize Model for Parameter Variations at Single Operating Point" on page 3-20

Application Description	Operating Point Computation Options	Linearization Workflow
Your model has multiple operating conditions that depend on the values of varying model parameters. Use this approach when creating linear time-varying (LTV) models.	• Batch trim your model for the varying parameter values, using a single model compilation when possible. Batch trimming is not supported in the Linear Analysis Tool. • Trim the model separately for each parameter value combination, which requires multiple model compilations. Use this option with the Linear Analysis Tool. • Compute an operating point at a simulation snapshot for each parameter value combination.	1 Define parameter values for trimming. 2 Compute operating points for the specified parameter value variations. 3 Batch linearize the model at the computed operating points using the corresponding parameter value combinations. For an example, see: • "Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations" on page 3-25 • "LPV Approximation of a Boost Converter Model" on page 3-117

In addition to varying operating points and model parameters, you can obtain multiple transfer functions from your system by varying the linearization I/O configuration using an `slLinearizer` interface. You can do so for a model with a single operating point and no parameter variation, and also for any of the batch linearization options in the preceding table. For more information, see "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41 and "Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32.

Choose Batch Linearization Tool

You can perform batch linearization using the **Linear Analysis Tool** or at the MATLAB command line using either the `linearize` function or an `slLinearizer` interface. Use the following table to choose a batch linearization tool.

Reasons to Use Linear Analysis Tool	Reasons to Use `linearize`	Reasons to Use `slLinearizer`
• You are new to Simulink Control Design software. • You have experience with the Linear Analysis Tool. • You do not want to batch trim your model, which is not supported in the Linear Analysis Tool.	• You are new to Simulink Control Design or have experience with Linear Analysis Tool, and you prefer to work at the command line or in a repeatable script. The workflow for using `linearize` closely mirrors the workflow for linearizing models using the Linear Analysis Tool. When you generate MATLAB code from the Linear Analysis Tool to reproduce your session programmatically, this code uses `linearize`. You can easily modify this code to batch linearize a model. • You are extracting linearizations for a single transfer function; that is, only one I/O set.	• You want to obtain multiple open-loop and closed-loop transfer functions without modifying the model or creating a linearization I/O set (using `linio`) for each transfer function. • You want to obtain multiple open-loop and closed-loop transfer functions without recompiling the model for each transfer function. You can also obtain multiple open-loop and closed-loop transfer functions using `linearize` or the Linear Analysis Tool. However, the software recompiles the model each time you change the I/O set.

See Also

More About

- "What Is Batch Linearization?" on page 3-2
- "Batch Linearization Efficiency When You Vary Parameter Values" on page 3-10
- "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75

- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-28
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-41

Batch Linearization Efficiency When You Vary Parameter Values

You can use the Simulink Control Design linearization tools to efficiently batch linearize a model at varying model parameter values. If all the model parameters you vary are tunable, the linearization tools use a single model compilation to compute linearizations for all parameter grid points.

Tunable and Nontunable Parameters

The term tunable parameters refers to parameters whose values you can change during model simulation without recompiling the model. In general, only parameters that represent mathematical variables are tunable. Common tunable parameters include the Gain parameter of the Gain block, PID gains of the PID Controller block, and Numerator and Denominator coefficients of the Transfer Fcn block.

In contrast, when you vary the value of nontunable parameters, the linearization tools compile the model for each parameter grid point. This repeated compilation makes batch linearization slower. Parameters that specify the appearance or structure of a block, such as the number of inputs of a Sum block, are not tunable. Parameters that specify when a block is evaluated, such as a block's sample time or priority, are also not tunable.

Controlling Model Recompilation

By default, the linearization tools compute all linearizations with a single compilation whenever it is possible to do so, i.e., whenever all parameters are tunable. If the software detects nontunable parameters specified for variation, it issues a warning and recompiles the model for each parameter-grid point. You can change this default behavior at the command line using the `AreParamsTunable` option of `linearizeOptions`. In the Linear Analysis Tool, click  **More Options** and use the **Recompile the model when parameter values are varied for linearization** option. The following table describes how these options affect the recompilation behavior.

	All varying parameters are tunable	Some varying parameters are not tunable
• Command line: `AreParamsTunable = true` (default) • Linear Analysis Tool: **Recompile the model when parameter values are varied for linearization** is unchecked (default)	Linearizations are computed for all parameter-grid points with a single compilation.	Model is recompiled for each parameter-grid point. Software issues a warning.
• Command line: `AreParamsTunable = false` • Linear Analysis Tool: **Recompile the model when parameter values are varied for linearization** is checked	Model is recompiled for each parameter-grid point.	Model is recompiled for each parameter-grid point. Warning is suppressed.

Suppose that you are performing batch linearization by varying the values of tunable parameters and notice that the software is recompiling the model more than necessary. To ensure that linearizations are computed with a single compilation whenever possible, make sure that:

- At the command line, the `AreParamsTunable` option is set to `true`.
- In Linear Analysis Tool, **Recompile the model when parameter values are varied for linearization** is unchecked.

See Also

`linearize` | `linearizeOptions` | `slLinearizer`

More About

- "Set Block Parameter Values" (Simulink)
- "Model Parameters" (Simulink)
- "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75
- "Batch Linearize Model for Parameter Variations at Single Operating Point" on page 3-20

- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32
- “Specify Parameter Samples for Batch Linearization” on page 3-62

Mark Signals of Interest for Batch Linearization

In this section...

When batch linearizing a model using an `slLinearizer` interface, you can mark signals of interest using analysis points. You can then analyze the response of your system at any of these points using functions such as `getIOTransfer` and `getLoopTransfer`.

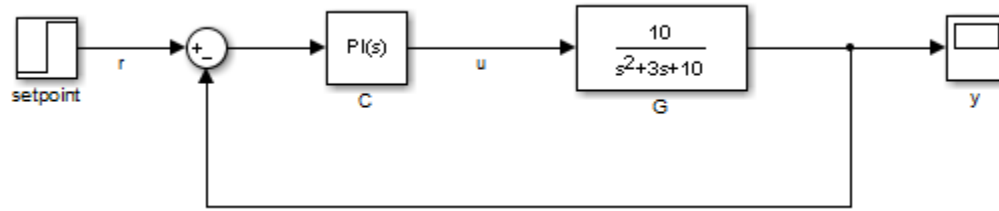
Alternatively, if you are batch linearizing your model using the:

- Linear Analysis Tool, specify analysis points as shown in “Specify Portion of Model to Linearize in Linear Analysis Tool” on page 2-29.
- `linearize` command, specify analysis points using `linio`.

For more information on selecting a batch linearization tool, see “Choose Batch Linearization Methods” on page 3-5.

Analysis Points

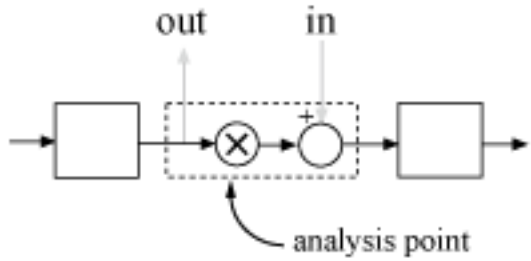
Analysis points identify locations within a Simulink model that are relevant for linear analysis. Each analysis point is associated with a signal that originates from the outport of a Simulink block. For example, in the following model, the reference signal `r` and the control signal `u` are analysis points that originate from the outputs of the setpoint and C blocks respectively.



Each analysis point can serve one or more of the following purposes:

- **Input** — The software injects an additive input signal at an analysis point, for example, to model a disturbance at the plant input.
- **Output** — The software measures the signal value at a point, for example, to study the impact of a disturbance on the plant output.
- **Loop Opening** — The software interprets a break in the signal flow at a point, for example, to study the open-loop response at the plant input.

When you use an analysis point for more than one purpose, the software applies the purposes in this sequence: output measurement, then loop opening, then input.



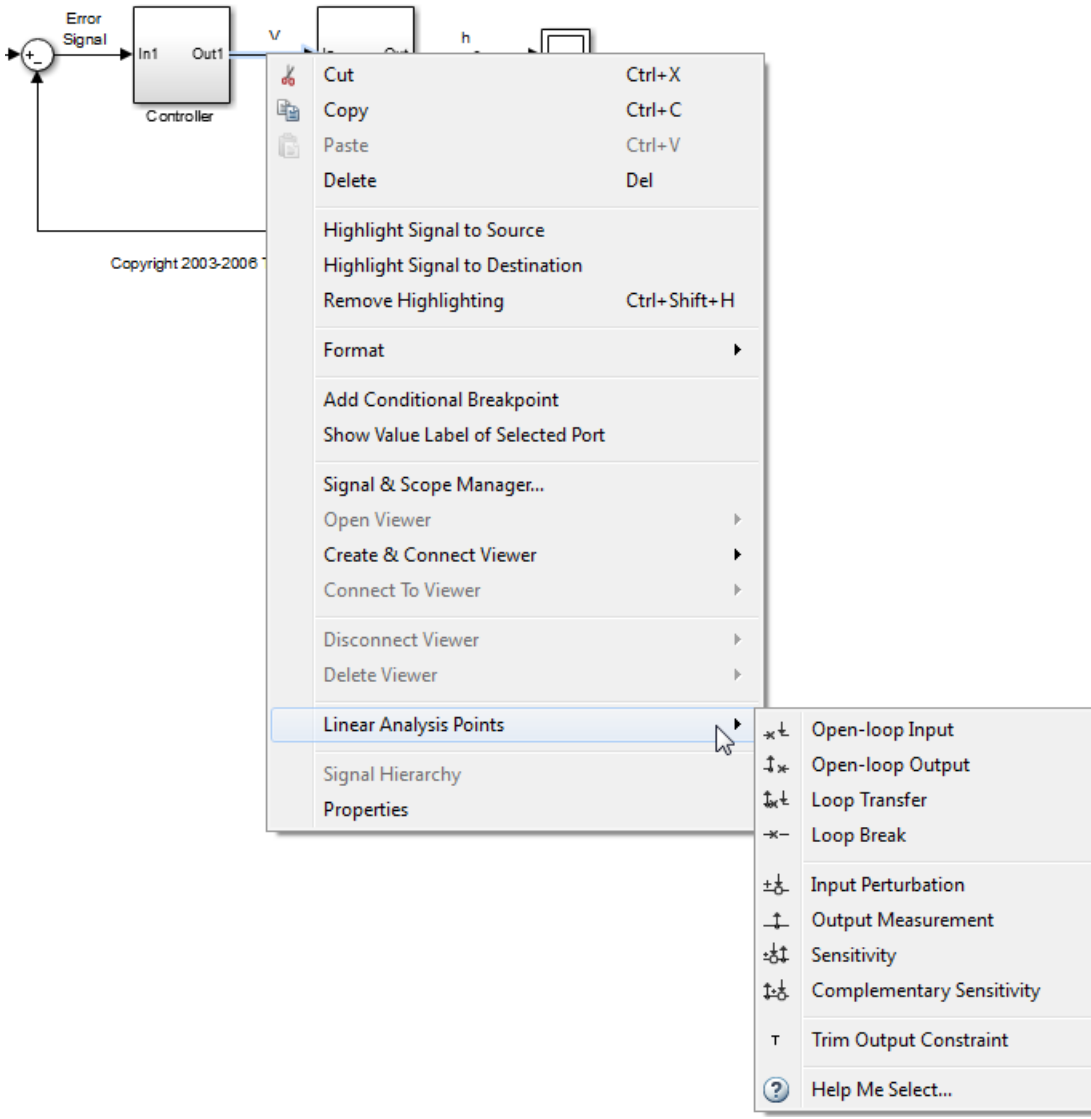
Using analysis points, you can extract open-loop and closed-loop responses from a Simulink model. You can also specify requirements for control system tuning using analysis points. For more information, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48.

Specify Analysis Points

You can mark analysis points either explicitly in the Simulink model, or programmatically using the `addPoint` command for an `slLinearizer` interface.

Mark Analysis Points in Simulink Model

To mark an analysis point explicitly in the model, right-click a signal and, under **Linear Analysis Points**, select an analysis point type.



You can select any of the following closed-loop analysis point types, which are equivalent within an `slLinearizer` interface.

- **Input Perturbation**

- **Output Measurement**
- **Sensitivity**
- **Complementary Sensitivity**

If you want to introduce a permanent loop opening at a signal as well, select one of the following open-loop analysis point types:

- **Open-Loop Input**
- **Open-Loop Output**
- **Loop Transfer**
- **Loop Break**

When you define a signal as an open-loop point, analysis functions such as `getIOTransfer` always enforce a loop break at that signal during linearization. All open-loop analysis point types are equivalent within an `slLinearizer` interface. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-39.

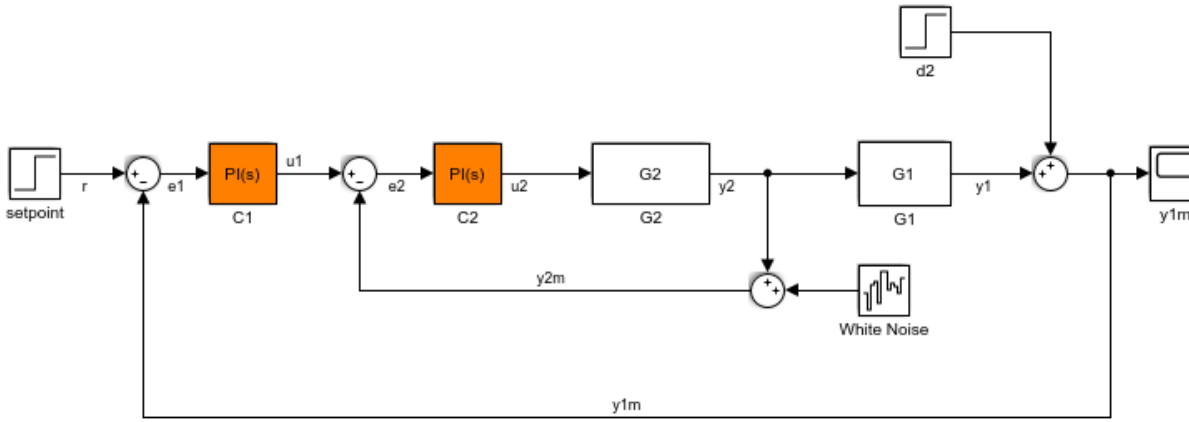
When you create an `slLinearizer` interface for a model, any analysis points defined in the model are automatically added to the interface. If you defined an analysis point using:

- A closed-loop type, the signal is added as an analysis point only.
- An open-loop type, the signal is added as both an analysis point and a permanent opening.

Mark Analysis Points Programmatically

To mark analysis points programmatically, use the `addPoint` command. For example, consider the `scdcascade` model.

```
open_system('scdcascade')
```



To mark analysis points, first create an `slLinearizer` interface.

```
sllin = slLinearizer('scdcascade');
```

To add a signal as an analysis point, use the `addPoint` command, specifying the source block and port number for the signal.

```
addPoint(sllin,'scdcascade/C1',1);
```

If the source block has a single output port, you can omit the port number.

```
addPoint(sllin,'scdcascade/G2');
```

For convenience, you can also mark analysis points using the:

- Name of the signal.

  ```
  addPoint(sllin,'y2');
  ```

- Combined source block path and port number.

  ```
  addPoint(sllin,'scdcascade/C1/1')
  ```

- End of the full source block path when unambiguous.

  ```
  addPoint(sllin,'G1/1')
  ```

You can also add permanent openings to an `slLinearizer` interface using the `addOpening` command, and specifying signals in the same way as for `addPoint`. For

more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-39.

```
addOpening(sllin,'y1m');
```

You can also define analysis points by creating linearization I/O objects using the `linio` command.

```
io(1) = linio('scdcascade/C1',1,'input');
io(2) = linio('scdcascade/G1',1,'output');
addPoint(sllin,io);
```

As when you define analysis points directly in your model, if you specify a linearization I/O object with:

- A closed-loop type, the signal is added as an analysis point only.
- An open-loop type, the signal is added as both an analysis point and a permanent opening.

Refer to Analysis Points

Once you have marked analysis points in an `slLinearizer` interface, you can analyze the response at any of these points using the following analysis functions:

- `getIOTransfer` — Transfer function for specified inputs and outputs
- `getLoopTransfer` — Open-loop transfer function from an additive input at a specified point to a measurement at the same point
- `getSensitivity` — Sensitivity function at a specified point
- `getCompSensitivity` — Complementary sensitivity function at a specified point

To view the available analysis points in an `slLinearizer` interface, use the `getPoints` command.

```
getPoints(sllin)


ans =

  3x1 cell array

    {'scdcascade/C1/1[u1]'}
```

```
    {'scdcascade/G2/1[y2]'}
    {'scdcascade/G1/1[y1]'}
```

To use an analysis point with an analysis function, you can specify an unambiguous abbreviation of the analysis point name returned by `getPoints`. For example, compute the transfer function from `u1` to `y1`, and find the sensitivity to a disturbance at the output of block G2.

```
ioSys = getIOTransfer(sllin,'u1','y1');
sensG2 = getSensitivity(sllin,'G2');
```

See Also

`addOpening` | `addPoint` | `getPoints` | `slLinearizer`

More About

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-48

Batch Linearize Model for Parameter Variations at Single Operating Point

In this example, you vary model parameters and linearize a model at its nominal operating conditions using the `linearize` command.

You can batch linearize a model for parameter variations at a single operating point to study:

- Plant dynamics for varying component tolerances.
- Controller robustness to variations in plant parameters.
- Transient responses for varying controller gains.

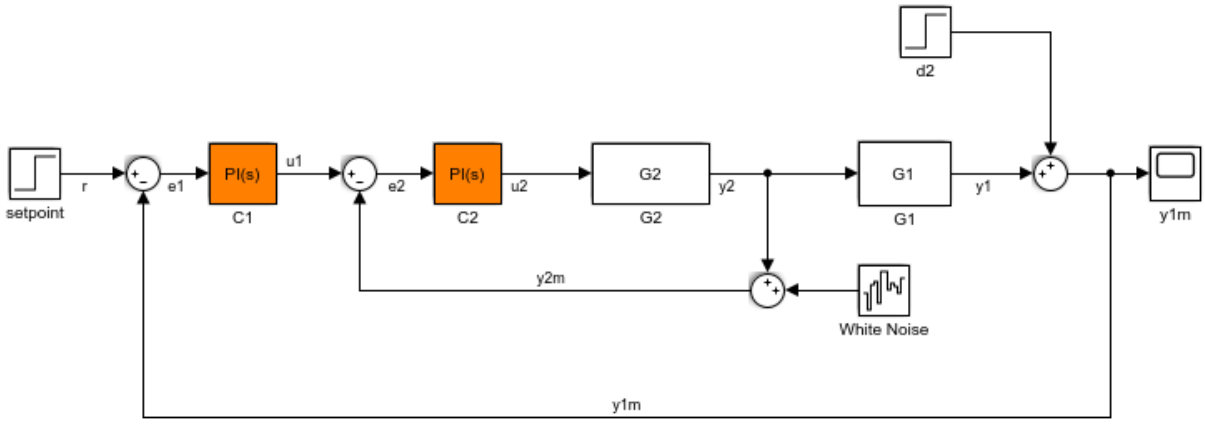
The `scdcascade` model contains two cascaded feedback control loops. Each loop includes a PI controller. The plant models, `G1` and `G2`, are LTI models.

For this model, the model operating point represents the nominal operating conditions of the system. Therefore, you do not have to trim the model before linearization. If your application includes parameter variations that affect the operating point of the model, you must first batch trim the model for the parameter variations. Then, you can linearize the model at the trimmed operating points. For more information, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-25.

To examine the effects of varying the outer-loop controller gains, linearize the model at the nominal operating point for each combination of gain values.

Open the model.

```
sys = 'scdcascade';
open_system(sys)
```



Define linearization input and output points for computing the closed-loop input/output response of the system.

```
io(1) = linio('scdcascade/setpoint',1,'input');
io(2) = linio('scdcascade/Sum',1,'output');
```

`io(1)`, the signal originating at the outport of the `setpoint` block, is the reference input. `io(2)`, the signal originating at the outport of the `Sum` block, is the system output.

To extract multiple open-loop and closed-loop transfer functions from the same model, batch linearize the system using an `slLinearizer` interface. For more information, see "Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32.

Vary the outer-loop controller gains, `Kp1` and `Ki1`, within 20% of their nominal values.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid,Ki1_grid] = ndgrid(Kp1_range,Ki1_range);
```

Create a parameter structure with fields `Name` and `Value`. `Name` indicates which the variable to vary in the model workspace, the MATLAB® workspace, or a data dictionary.

```
params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;
```

`params` is a 6-by-4 parameter value grid, where each grid point corresponds to a unique combination of `Kp1` and `Ki1` values.

Obtain the closed-loop transfer function from the reference input to the plant output for the specified parameter values. If you do not specify an operating point, `linearize` uses the current model operating point.

```
G = linearize(sys,io,params);
```

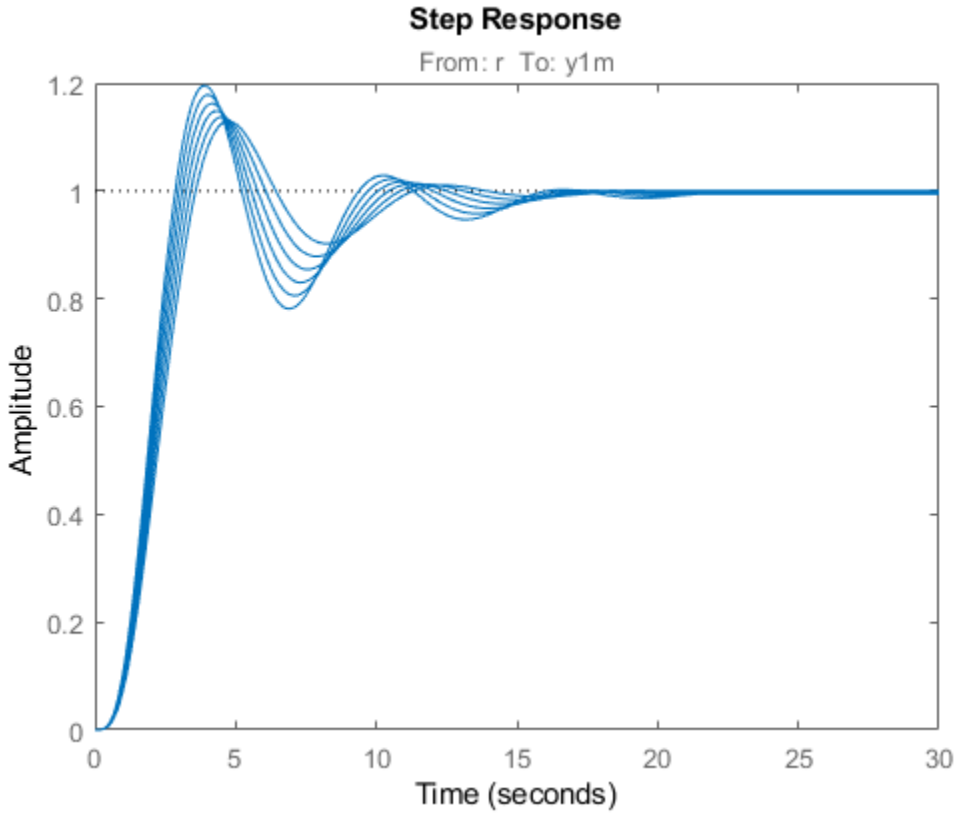
`G` is a 6-by-4 array of linearized models. Each entry in the array contains a linearization for the corresponding parameter combination in `params`. For example, `G(:,:,2,3)` corresponds to the linearization obtained by setting the values of the `Kp1` and `Ki1` parameters to `Kp1_grid(2,3)` and `Ki1_grid(2,3)`, respectively. The set of parameter values corresponding to each entry in the model array `G` is stored in the `SamplingGrid` property of `G`. For example, examine the corresponding parameter values for linearization `G(:,:,2,3)`:

```
G(:,:,2,3).SamplingGrid
```

```
ans =

  struct with fields:

    Kp1: 0.1386
    Ki1: 0.0448
```

To study the effects of the varying gain values, analyze the linearized models in `G`. For example, examine the step responses for all `Kp2` values and the third `Ki1` value.

```
stepplot(G(:,:,:,3))
```



See Also

`linearize` | `linio` | `ndgrid`

More About

- “watertank Simulink Model”
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10
- “Specify Parameter Samples for Batch Linearization” on page 3-62
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-48

- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-28
- “Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool” on page 3-75
- “LPV Approximation of a Boost Converter Model” on page 3-117

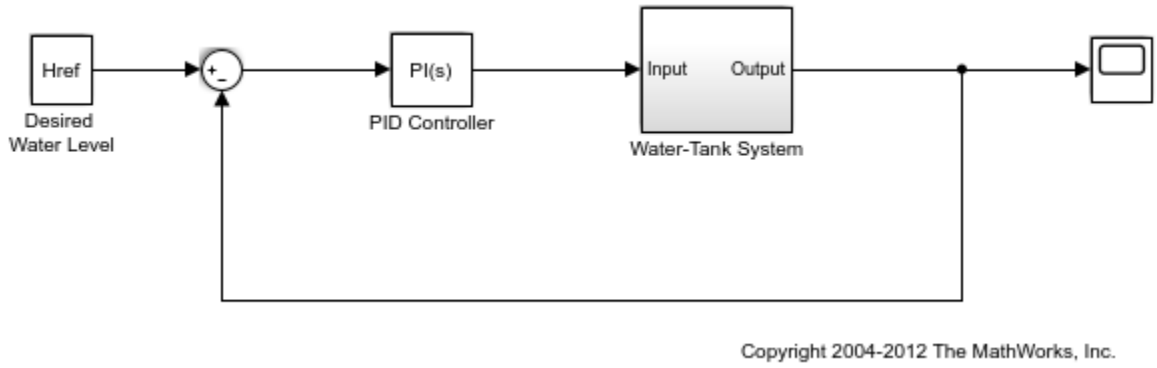
Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations

If your application includes parameter variations that affect the operating point of the model, you must batch trim the model for the parameter variations before linearization. Use this batch linearization approach when computing linear models for linear parameter-varying systems.

For more information on batch trimming models for parameter variations, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-65.

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Vary parameters `A` and `b` within 10% of their nominal values. Specify three values for `A` and four values for `b`, creating a 3-by-4 value grid for each parameter.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
```

```
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the mode that specifies that both model states are unknown and must be at steady state in the trimmed operating point.

```
opspec = operspec(sys);
```

Trim the model using the specified operating point specification, parameter grid, and option set. Suppress the display of the operating point search report.

```
opt = findopOptions('DisplayReport','off');
[op,opreport] = findop(sys,opspec,params,opt);
```

`findop` trims the model for each parameter combination using only one model compilation. `op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

To compute the closed-loop input/output transfer function for the model, define the linearization input and output points as the reference input and model output, respectively.

```
io(1) = linio('watertank/Desired  Water Level',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

To extract multiple open-loop and closed-loop transfer functions from the same model, batch linearize the system using an `slLinearizer` interface. For more information, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32.

Batch linearize the model at the trimmed operating points using the specified I/O points and parameter variations.

```
G = linearize(sys,op,io,params);
```

`G` is a 3-by-4 array of linearized models. Each entry in the array contains a linearization for the corresponding parameter combination in `params`. For example, `G(:,:,2,3)` corresponds to the linearization obtained by setting the values of the `A` and `b` parameters to `A_grid(2,3)` and `b_grid(2,3)`, respectively. The set of parameter values corresponding to each entry in the model array `G` is stored in the `SamplingGrid` property of `G`. For example, examine the corresponding parameter values for linearization `G(:,:,2,3)`:

```
G(:,:,2,3).SamplingGrid
```

```
ans =

  struct with fields:

    A: 20
    b: 5.1667
```

When batch linearizing for parameter variations, you can obtain the linearization offsets that correspond to the linearization operating points. To do so, set the `StoreOffsets` linearization option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Linearize the model using the specified parameter grid, and return the linearization offsets in the `info` structure.

```
[G,~,info] = linearize('watertank',io,params,opt);
```

You can then use the offsets to configure an LPV System block. To do so, you must first convert the offsets to the required format. For an example, see "LPV Approximation of a Boost Converter Model" on page 3-117.

```
offsets = getOffsetsForLPV(info);
```

See Also

`findop` | `linearize` | `linio` | `ndgrid`

More About

- "Batch Linearization Efficiency When You Vary Parameter Values" on page 3-10
- "Specify Parameter Samples for Batch Linearization" on page 3-62
- "Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48
- "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75
- "LPV Approximation of a Boost Converter Model" on page 3-117

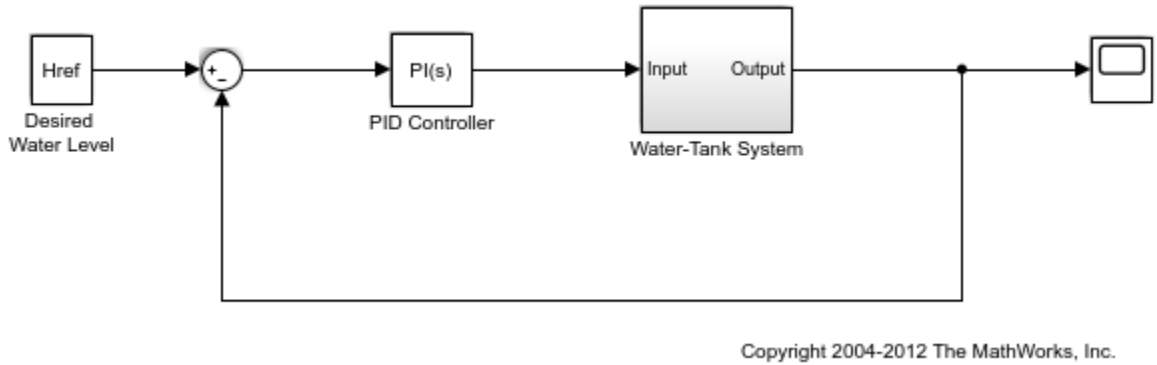
Batch Linearize Model at Multiple Operating Points Using linearize Command

This example shows how to use the `linearize` command to batch linearize a model at varying operating points.

Obtain the plant transfer function, modeled by the Water-Tank System block, for the `watertank` model. You can analyze the batch linearization results to study the operating point effects on the model behavior.

Open the model.

```
open_system('watertank')
```



Specify the linearization I/Os.

```
ios(1) = linio('watertank/PID Controller',1,'input');
ios(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

`ios(2)` specifies an open-loop output point; the loop opening eliminates the effects of feedback.

You can linearize the model using trimmed operating points, the model initial condition, or simulation snapshot times. For this example, linearize the model at specified simulation snapshot times.

```
ops_tsnapshot = [1,20];
```

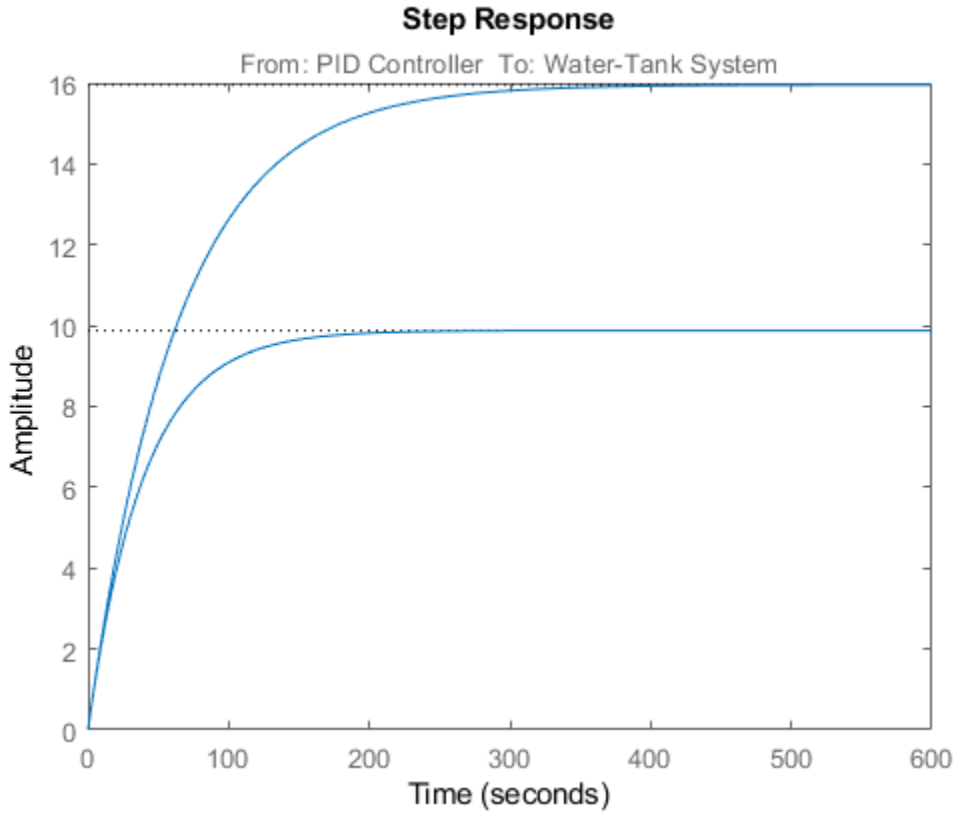
Obtain the transfer function for the Water-Tank System block, linearizing the model at the specified operating points.

```
T = linearize('watertank',ios,ops_tsnapshot);
```

`T` is a 2 x 1 array of linearized continuous-time state-space models. The software computes the `T(:,:,1)` model by linearizing `watertank` at `ops_tsnapshot(1)`, and `T(:,:,2)` by linearizing `watertank` at `ops_tsnapshot(2)`.

Use Control System Toolbox analysis commands to examine the properties of the linearized models in `T`. For example, examine the step response of the plant at both snapshot times.

```
stepplot(T)
```



See Also

`findop` | `linearize` | `linio` | `stepplot`

More About

- “watertank Simulink Model”
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20

- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-48

Vary Parameter Values and Obtain Multiple Transfer Functions

This example shows how to use the `slLinearizer` interface to batch linearize a Simulink® model. You vary model parameter values and obtain multiple open-loop and closed-loop transfer functions from the model.

You can perform the same analysis using the `linearize` command. However, when you want to obtain multiple open-loop and closed-loop transfer functions, especially for models that are expensive to compile repeatedly, `slLinearizer` can be more efficient.

Since the parameter variations in this example do not affect the operating point of the model, you batch linearize the model at a single operating point. If your application uses parameter variations that affect the model operating point, first trim the model for each parameter value combination. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-25.

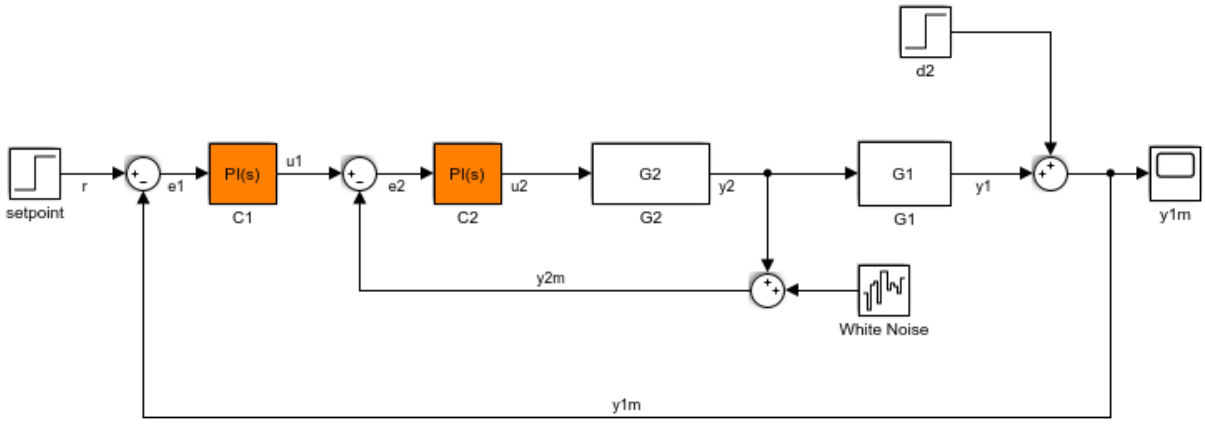
Create slLinearizer Interface for Model

The `scdcascade` model used for this example contains a pair of cascaded feedback control loops. Each loop includes a PI controller. The plant models, `G1` (outer loop) and `G2` (inner loop), are LTI models.

Use the `slLinearizer` interface to analyze the inner-loop and outer-loop dynamics.

Open the model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Use the `slLinearizer` command to create the interface.

```
sllin = slLinearizer(mdl)
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters          : []
      OperatingPoints     : [] (model initial condition will be used.)
      BlockSubstitutions  : []
      Options             : [1x1 linearize.LinearizeOptions]
```

The Command Window display shows information about the `slLinearizer` interface. In this interface, no parameters to vary are yet specified, so the `Paramaeters` property is empty.

Vary Inner-Loop Controller Gains

For inner-loop analysis, vary the gains of the inner-loop PI controller block, C2. Vary the proportional gain (`Kp2`) and integral gain (`Ki2`) in the 15% range.

```
Kp2_range = linspace(Kp2*0.85,Kp2*1.15,6);
Ki2_range = linspace(Ki2*0.85,Ki2*1.15,4);
[Kp2_grid, Ki2_grid] = ndgrid(Kp2_range,Ki2_range);
```

```
params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;
params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

`Kp2_range` and `Ki2_range` specify the sample values for `Kp2` and `Ki2`. To obtain a transfer function for each combination of `Kp2` and `Ki2`, use `ndgrid` and create a 6 x 4 parameter grid with grid arrays `Kp2_grid` and `Ki2_grid`. Configure the `Parameters` property of `sllin` with the structure `params`. This structure specifies the parameters to be varied and their grid arrays.

Analyze Closed-Loop Transfer Function for the Inner Loop

The overall closed-loop transfer function for the inner loop is equal to the transfer function from `u1` to `y2`. To eliminate the effects of the outer loop, you can break the loop at `e1`, `y1m`, or `y1`. For this example, break the loop at `e1`.

Add `u1` and `y2` as analysis points, and `e1` as a permanent opening of `sllin`.

```
addPoint(sllin,{'y2','u1'});
addOpening(sllin,'e1');
```

Obtain the transfer function from `u1` to `y2`.

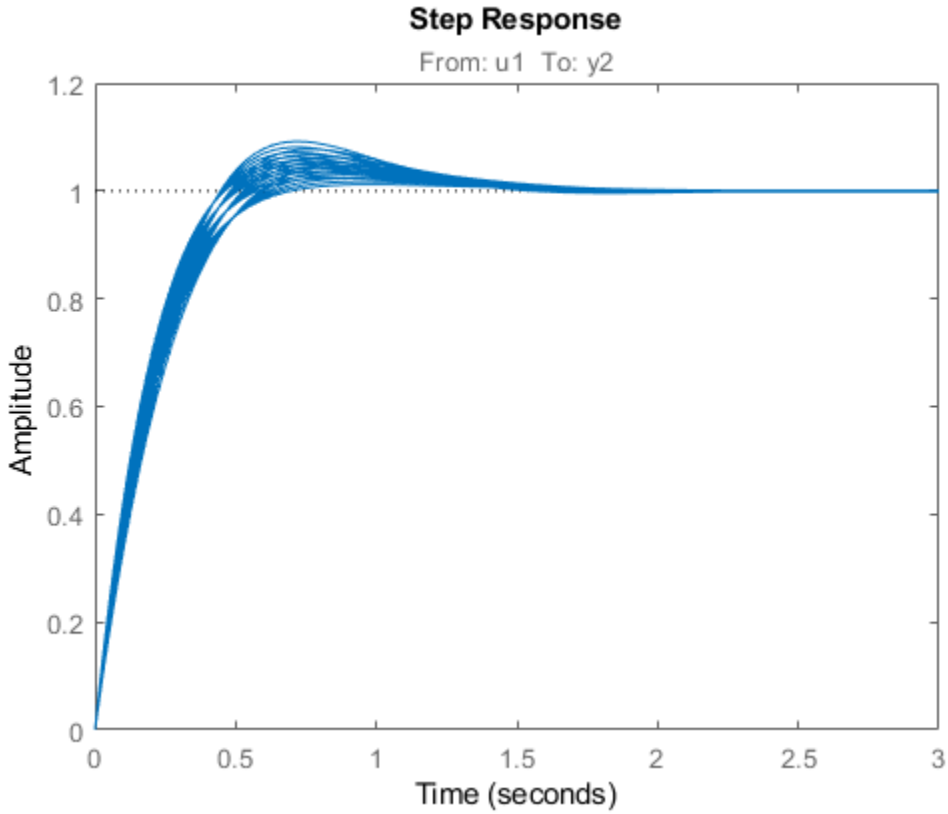
```
r2yi = getIOTransfer(sllin,'u1','y2');
```

`r2yi`, a 6 x 4 state-space model array, contains the transfer function for each specified parameter combination. The software uses the model initial conditions as the linearization operating point.

Because `e1` is a permanent opening of `sllin`, `r2yi` does not include the effects of the outer loop.

Plot the step response for `r2yi`.

```
stepplot(r2yi);
```



The step response for all models varies in the 10% range and the settling time is less than 1.5 seconds.

Analyze Inner-Loop Transfer Function at the Plant Output

Obtain the inner-loop transfer function at `y2`, with the outer loop open at `e1`.

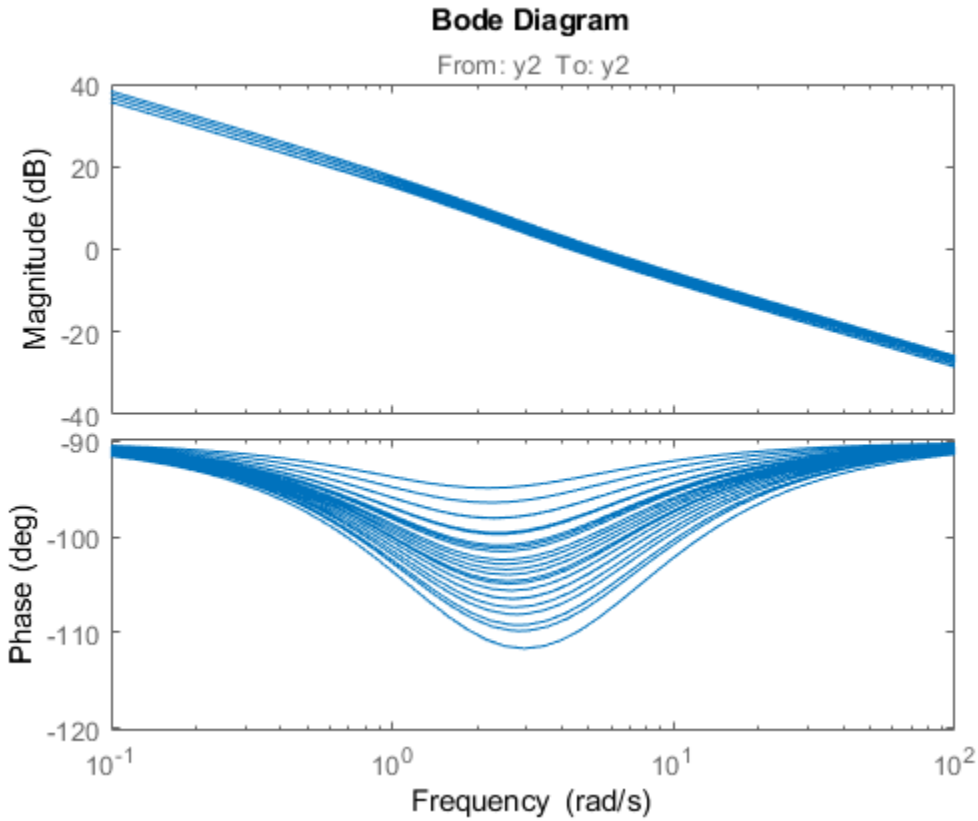
```
Li = getLoopTransfer(sllin,'y2',-1);
```

Because the software assumes positive feedback by default and `scdcascade` uses negative feedback, specify the feedback sign using the third input argument. Now, $L_i = -G_2C_2$. The `getLoopTransfer` command returns an array of state-space (`ss`)

models, one for each entry in the parameter grid. The `SamplingGrid` property of `Li` matches the parameter values with the corresponding `ss` model.

Plot the bode response for L_i.

```
bodeplot(Li);
```



The magnitude plot for all the models varies in the 3-dB range. The phase plot shows the most variation, approximately 20°, in the `[1 10]` rad/s interval.

Vary Outer-Loop Controller Gains

For outer-loop analysis, vary the gains of the outer-loop PI controller block, C1. Vary the proportional gain (`Kp1`) and integral gain (`Ki1`) in the 20% range.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid, Ki1_grid] = ndgrid(Kp1_range,Ki1_range);

params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;

sllin.Parameters = params;
```

Similar to the workflow for configuring the parameter grid for inner-loop analysis, create the structure, `params`, that specifies a 6 x 4 parameter grid. Reconfigure `sllin.Parameters` to use the new parameter grid. `sllin` now uses the default values for `Kp2` and `Ki2`.

Analyze Closed-Loop Transfer Function from Reference to Plant Output

Remove `e1` from the list of permanent openings for `sllin` before proceeding with outer-loop analysis.

```
removeOpening(sllin,'e1');
```

To obtain the closed-loop transfer function from the reference signal, `r`, to the plant output, `y1m`, add `r` and `y1m` as analysis points to `sllin`.

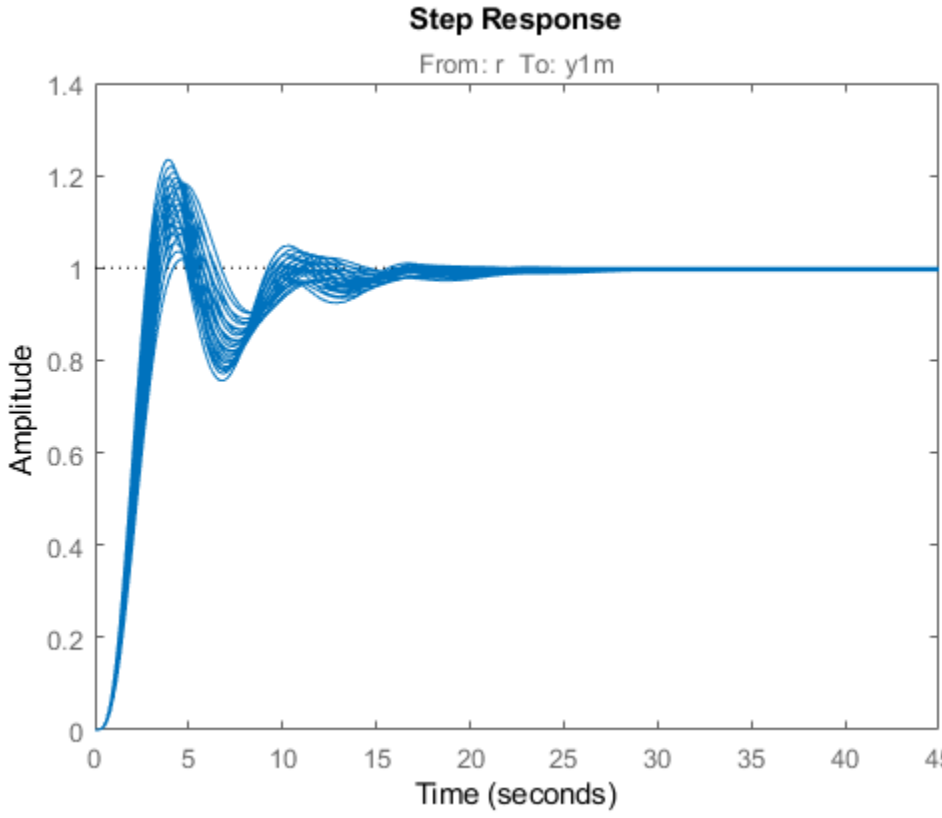
```
addPoint(sllin,{'r','y1m'});
```

Obtain the transfer function from `r` to `y1m`.

```
r2yo = getIOTransfer(sllin,'r','y1m');
```

Plot the step response for `r2yo`.

```
stepplot(r2yo);
```



The step response is underdamped for all the models.

Analyze Outer-Loop Sensitivity at Plant Output

To obtain the outer-loop sensitivity at the plant output, add `y1` as an analysis point to `sllin`.

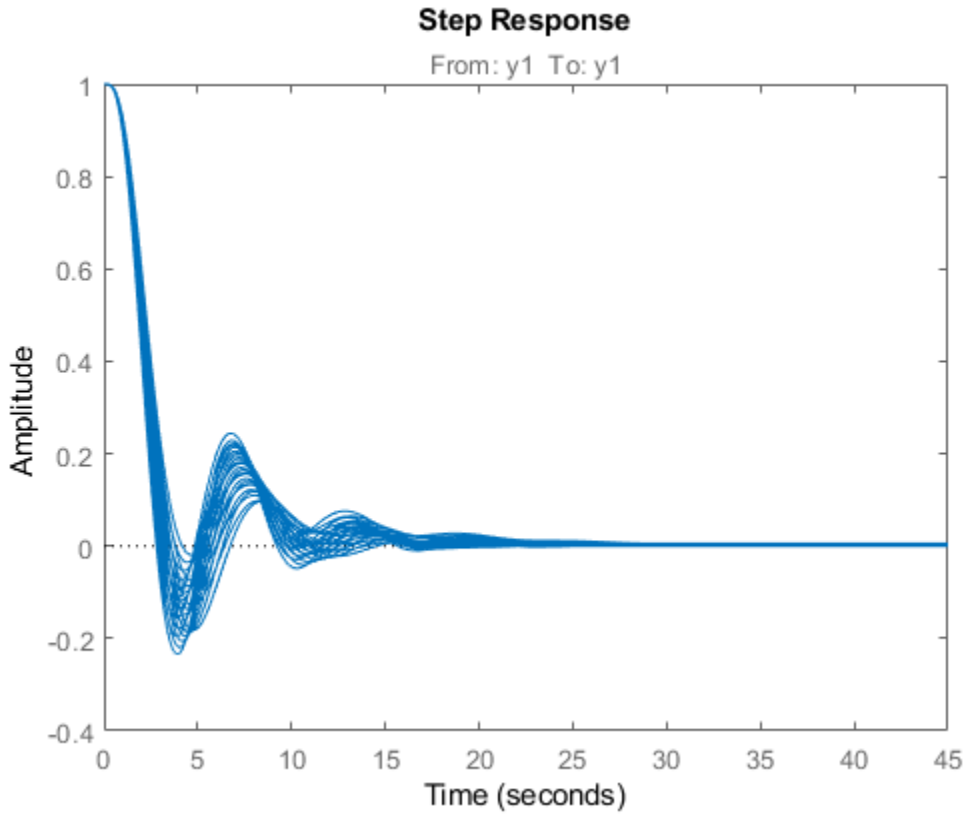
```
addPoint(sllin,'y1');
```

Obtain the outer-loop sensitivity at `y1`.

```
So = getSensitivity(sllin,'y1');
```

Plot the step response of `So`.

```
stepplot(So)
```



The plot indicates that it takes approximately 15 seconds to reject a step disturbance at the plant output, `y1`.

Obtain Linearization Offsets

When batch linearizing for parameter variations, you can obtain the linearization offsets that correspond to the linearization operating points. To do so, set the `StoreOffsets` linearization option in the `slLinearizer` interface.

```
sllin.Options.StoreOffsets = true;
```

When you call a linearization function using `sllin`, you can return linearization offsets in the `info` structure.

```
[r2yi,info] = getIOTransfer(sllin,'u1','y2');
```

You can then use the offsets to configure an LPV System block. To do so, you must first convert the offsets to the required format. For an example that uses the `linearize` command, see “LPV Approximation of a Boost Converter Model” on page 3-117.

```
offsets = getOffsetsForLPV(info);
```

Close the model.

```
bdclose(mdl);
```

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize` | `slLinearizer`

More About

- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10
- “Specify Parameter Samples for Batch Linearization” on page 3-62
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-48
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-41
- “Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool” on page 3-75

Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface

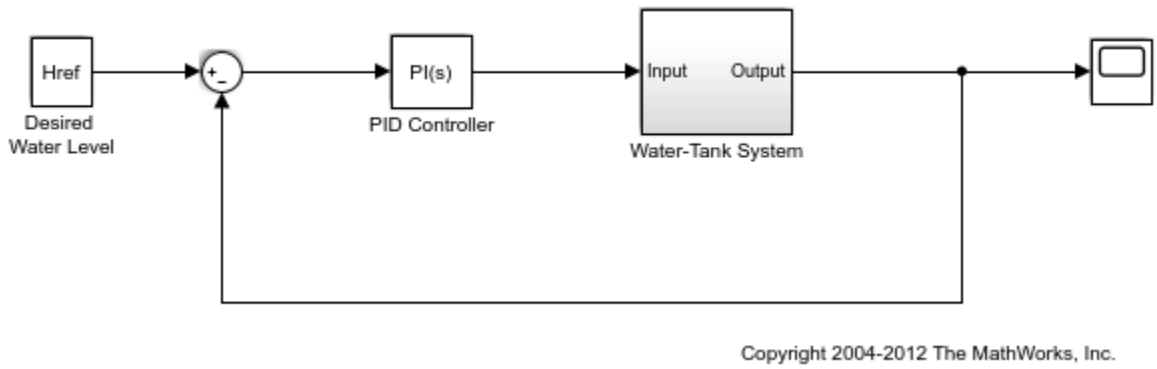
This example shows how to use the `slLinearizer` interface to batch linearize a Simulink® model. You linearize a model at multiple operating points and obtain multiple open-loop and closed-loop transfer functions from the model.

You can perform the same analysis using the `linearize` command. However, when you want to obtain multiple open-loop and closed-loop transfer functions, especially for models that are expensive to compile repeatedly, `slLinearizer` can be more efficient.

Create slLinearizer Interface for Model

Open the model.

```
mdl = 'watertank';
open_system(mdl);
```



Use the `slLinearizer` command to create the interface.

```
sllin = slLinearizer(mdl)
```

```
slLinearizer linearization interface for "watertank":

No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
```

```
Parameters         : []
OperatingPoints    : [] (model initial condition will be used.)
BlockSubstitutions : []
Options            : [1x1 linearize.LinearizeOptions]
```

The command-window display shows information about the `slLinearizer` interface. In this interface, the `OperatingPoints` property display shows that no operating point is specified.

Specify Multiple Operating Points for Linearization

You can linearize the model using trimmed operating points, the model initial condition, or simulation snapshot times. For this example, use trim points that you obtain for varying water-level reference heights.

```
opspec = operspec(mdl);
opspec.States(2).Known = 1;
opts = findopOptions('DisplayReport','off');

h = [10 15 20];

for ct = 1:numel(h)
    opspec.States(2).x = h(ct);
    Href = h(ct);
    ops(ct) = findop(mdl,opspec,opts);
end

sllin.OperatingPoints = ops;
```

Here, `h` specifies the different water-levels. `ops` is a 1 x 3 array of operating point objects. Each entry of `ops` is the model operating point at the corresponding water level. Configure the `OperatingPoints` property of `sllin` with `ops`. Now, when you obtain transfer functions from `sllin` using the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` functions, the software returns a linearization for each specified operating point.

Each trim point is only valid for the corresponding reference height, represented by the `Href` parameter of the Desired Water Level block. So, configure `sllin` to vary this parameter accordingly.

```
param.Name = 'Href';
param.Value = h;

sllin.Parameters = param;
```

Analyze Plant Transfer Function

In the `watertank` model, the Water-Tank System block represents the plant. To obtain the plant transfer function, add the input and output signals of the Water-Tank System block as analysis points of `sllin`.

```
addPoint(sllin,{'watertank/PID Controller','watertank/Water-Tank System'})
sllin
```

```
slLinearizer linearization interface for "watertank":

2 Analysis points:
--------------------------
Point 1:
- Block: watertank/PID Controller
- Port: 1
Point 2:
- Block: watertank/Water-Tank System
- Port: 1

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : [1x1 struct], 1 parameters with sampling grid of size 1x3
            "Href", varying between 10 and 20.
      OperatingPoints    : [1x3 opcond.OperatingPoint]
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The first analysis point, which originates at the outport of the PID Controller block, is the input to the Water-Tank System block. The second analysis point is the output of the Water-Tank System block.

Obtain the plant transfer function from the input of the Water-Tank System block to the block output. To eliminate the effects of the feedback loop, specify the block output as a temporary loop opening.

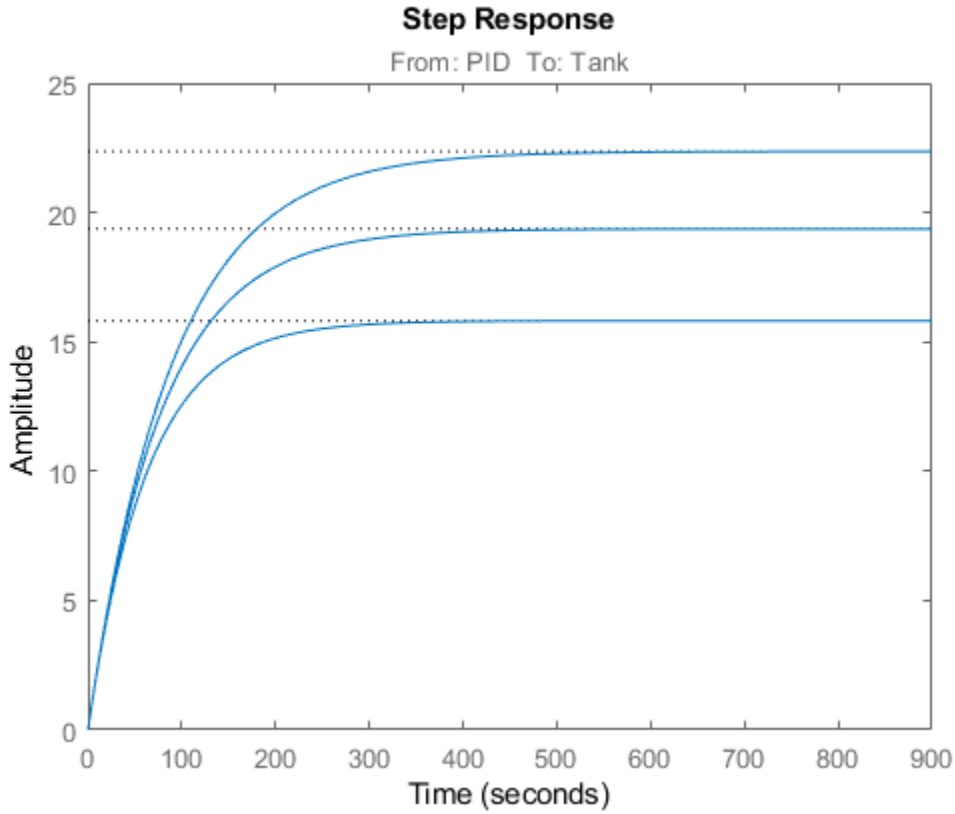
```
G = getIOTransfer(sllin,'PID','Tank','Tank');
```

In the call to `getIOTransfer`, `'PID'`, a portion of the block name `'watertank/PID Controller'`, specifies the first analysis point as the transfer function input. Similarly, `'Tank'`, a portion of the block name `'watertank/Water-Tank System'`, refers to the second analysis point. This analysis point is specified as the transfer function output (third input argument) and a temporary loop opening (fourth input argument).

The output, `G`, is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `G`.

```
stepplot(G);
```



The step response of the plant models varies significantly at the different operating points.

Analyze Closed-Loop Transfer Function

The closed-loop transfer function is equal to the transfer function from the reference input, originating at the Desired Water Level block, to the plant output.

Add the reference input signal as an analysis point of `sllin`.

```
addPoint(sllin,'watertank/Desired  Water Level');
```

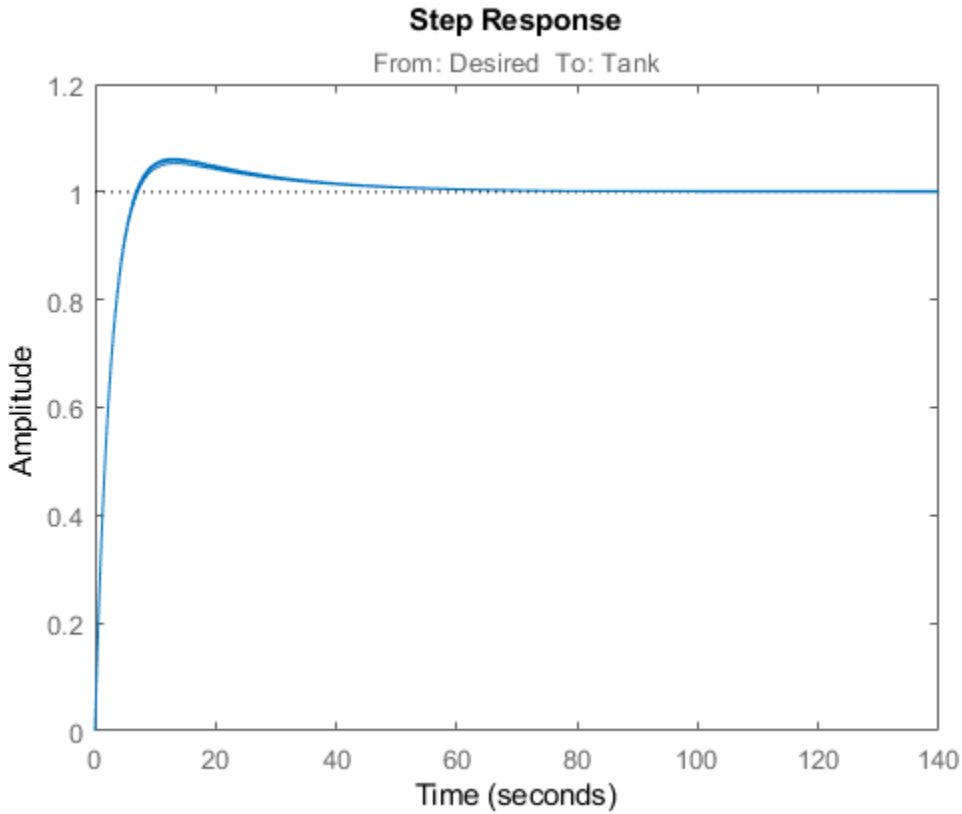
Obtain the closed-loop transfer function.

```
T = getIOTransfer(sllin,'Desired','Tank');
```

The output, `T`, is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `T`.

```
stepplot(T);
```



Although the step response of the plant transfer function varies significantly at the three trimmed operating points, the controller brings the closed-loop responses much closer together at all three operating points.

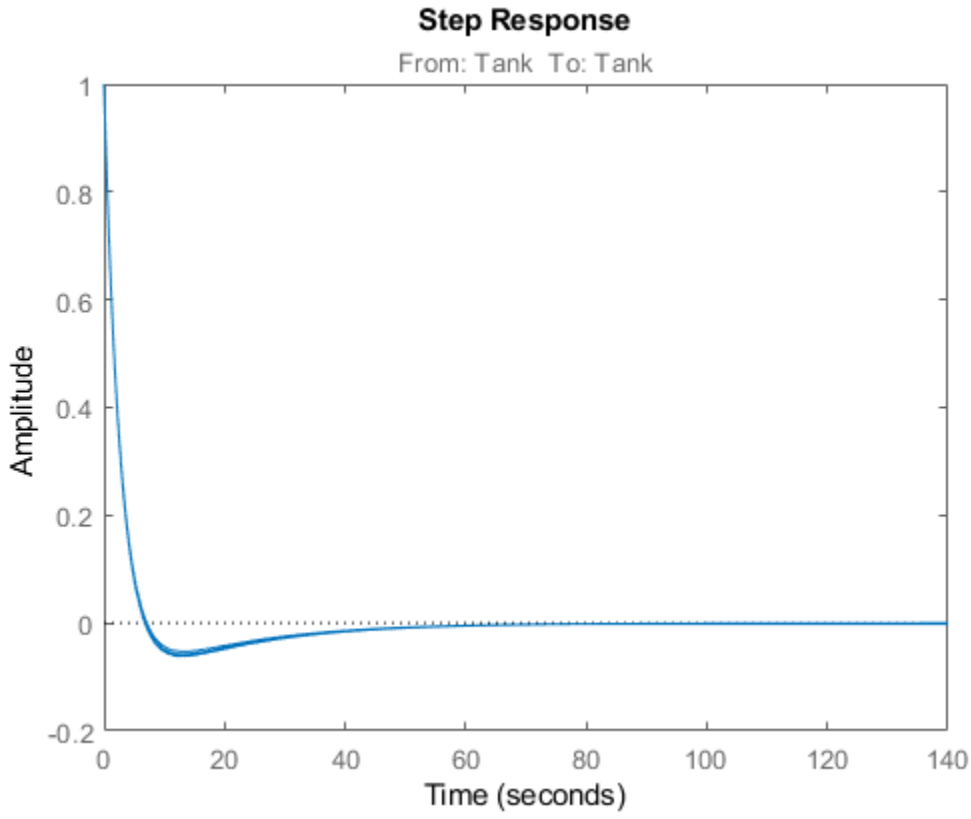
Analyze Sensitivity at Plant Output

```
S = getSensitivity(sllin,'Tank');
```

The software injects a disturbance signal and measures the output at the plant output. `S` is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `S`.

```
stepplot(S);
```



The plot indicates that both models can reject a step disturbance at the plant output within 40 seconds.

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize` | `slLinearizer`

More About

- "watertank Simulink Model"
- "Batch Compute Steady-State Operating Points for Multiple Specifications" on page 1-61
- "Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32
- "Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48

Analyze Command-Line Batch Linearization Results Using Response Plots

This example shows how to plot and analyze the step response for batch linearization results obtained at the command line. The term batch linearization results refers to the `ss` model array returned by the `slLinearizer` interface or `linearize` function. This array contains linearizations for varying parameter values, operating points, or both, such as illustrated in "Batch Linearize Model for Parameter Variations at Single Operating Point" on page 3-20 and "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41. You can use the techniques illustrated in this example to analyze the frequency response, stability, or sensitivity for batch linearization results.

Obtain Batch Linearization Results

Load the batch linearization results saved in `scd_batch_lin_results1.mat`.

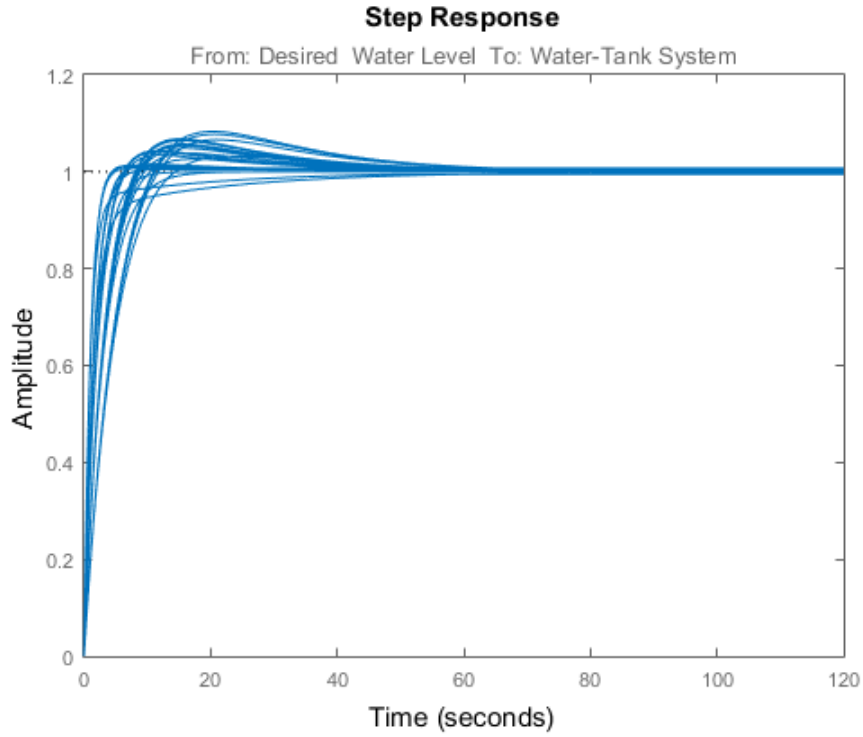
The following code obtains linearizations of the `watertank` model for four simulation snapshot times, `t = [0 1 2 3]`. At each snapshot time, the model parameters, `A` and `b`, are varied. The sample values for `A` are `[10 20 30]`, and the sample values for `b` are `[4 6]`. The `slLinearizer` interface includes analysis points at the reference signal and plant output.

```
open_system('watertank')
sllin = slLinearizer('watertank',{'watertank/Desired  Water Level',...
                                  'watertank/Water-Tank System'})

[A_grid,b_grid] = ndgrid([10,20,30],[4 6]);
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;

sllin.Parameters = params;
sllin.OperatingPoints = [0,1,2,3];

linsys = getIOTransfer(sllin,'Desired  Water Level','Water-Tank System');
```

`linsys`, a 4-by-3-by-2 `ss` model array, contains the closed-loop transfer function of the linearized `watertank` model from the reference input to the plant output. The operating point varies along the first array dimension of `linsys`, and the parameters `A` and `b` vary along the second and third dimensions, respectively.

Plot Step Responses of the Linearized Models

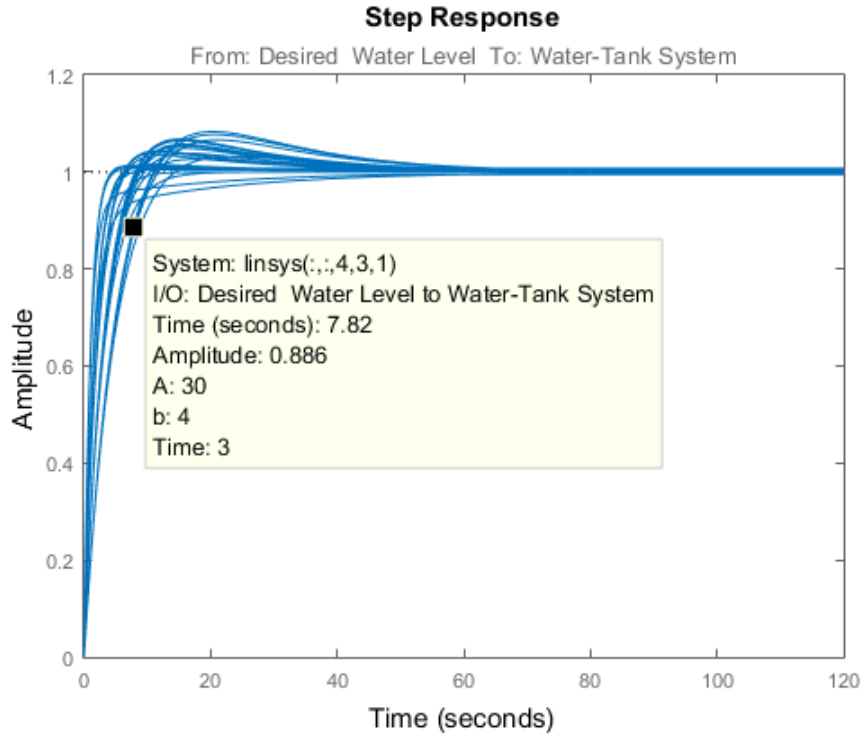
```
stepplot(linsys)
```



The step plot shows the responses of every model in the array. This plot shows the range of step responses of the system in the operating ranges covered by the parameter grid and snapshot times.

View Parameters and Snapshot Time of a Response

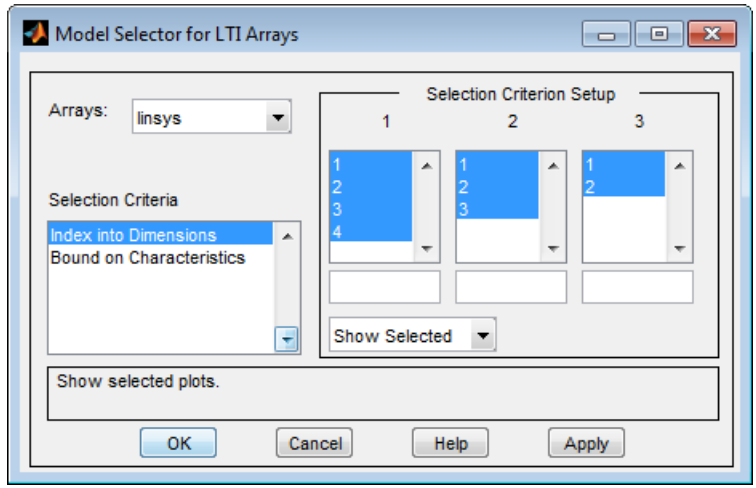
To view the parameters associated with a particular response, click the response on the plot.



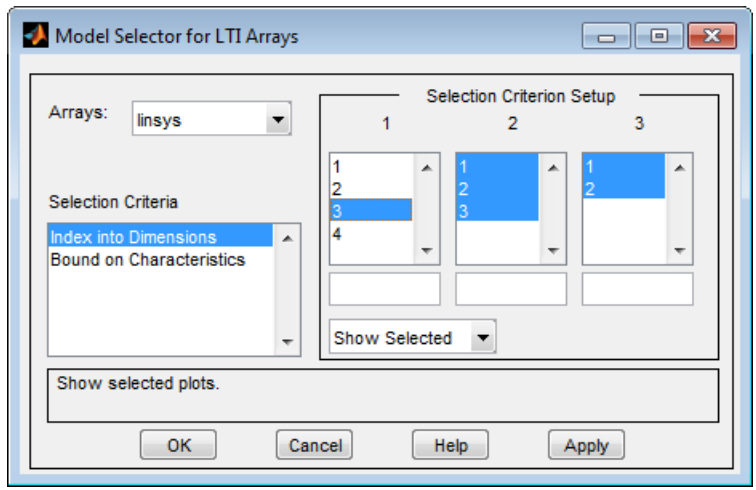
A data tip appears on the plot, providing information about the selected response and the related model. The last lines of the data tip show the parameter combination and simulation snapshot time that yielded this response. For example, in this previous plot, the selected response corresponds to the model obtained by setting `A` to `30` and `b` to `4`. The software linearized the model after simulating the model for three time units.

View Step Response of Subset of Results

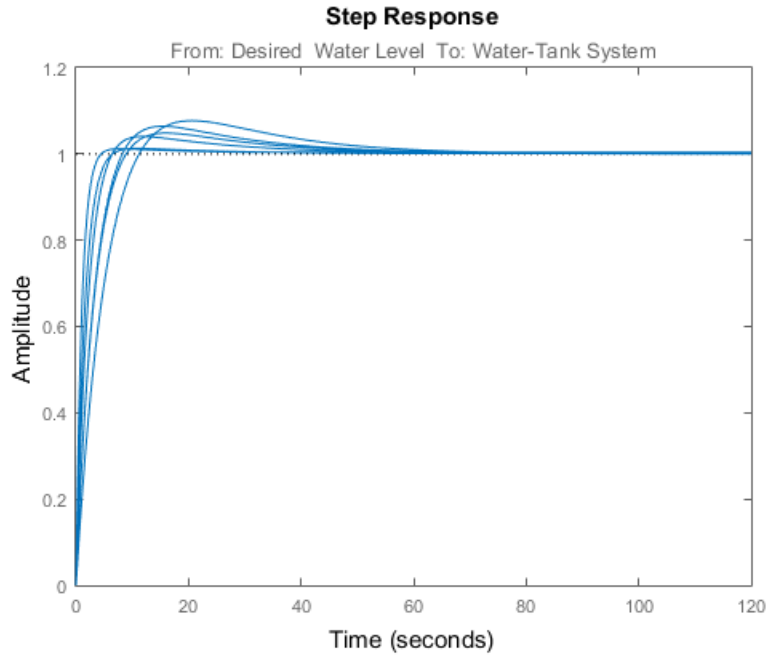
Suppose you want to view the responses for models linearized at a specific simulation snapshot time, such as two time units. Right-click the plot and select **Array Selector**. The Model Selector for LTI Arrays dialog box opens.



The **Selection Criterion Setup** panel contains three columns, one for each model array dimension of `linsys`. The first column corresponds to the simulation snapshot time. The third entry of this column corresponds to the simulation snapshot time of two time units, because the snapshot time array was `[0,1,2,3]`. Select only this entry in the first column.



Click **OK**. The plot displays the responses for only the models linearized at two time units.



Plot Step Response for Specific Parameter Combination and Snapshot Time

Suppose you want to examine only the step response for the model obtained by linearizing the `watertank` model at `t = 3`, for `A = 10` and `b = 4`. To do so, you can use the `SamplingGrid` property of `linsys`, which is specified as a structure. When you perform batch linearization, the software populates `SamplingGrid` with information regarding the variable values used to obtain the model. The variable values include each parameter that you vary and the simulation snapshot times at which you linearize the model. For example:

```
linsys(:,:,1).SamplingGrid

ans =

       A: 10
       b: 4
    Time: 0
```

Here `linsys(:,:,1)` refers to the first model in `linsys`. This model was obtained at simulation time `t = 0`, for `A = 10` and `b = 4`.

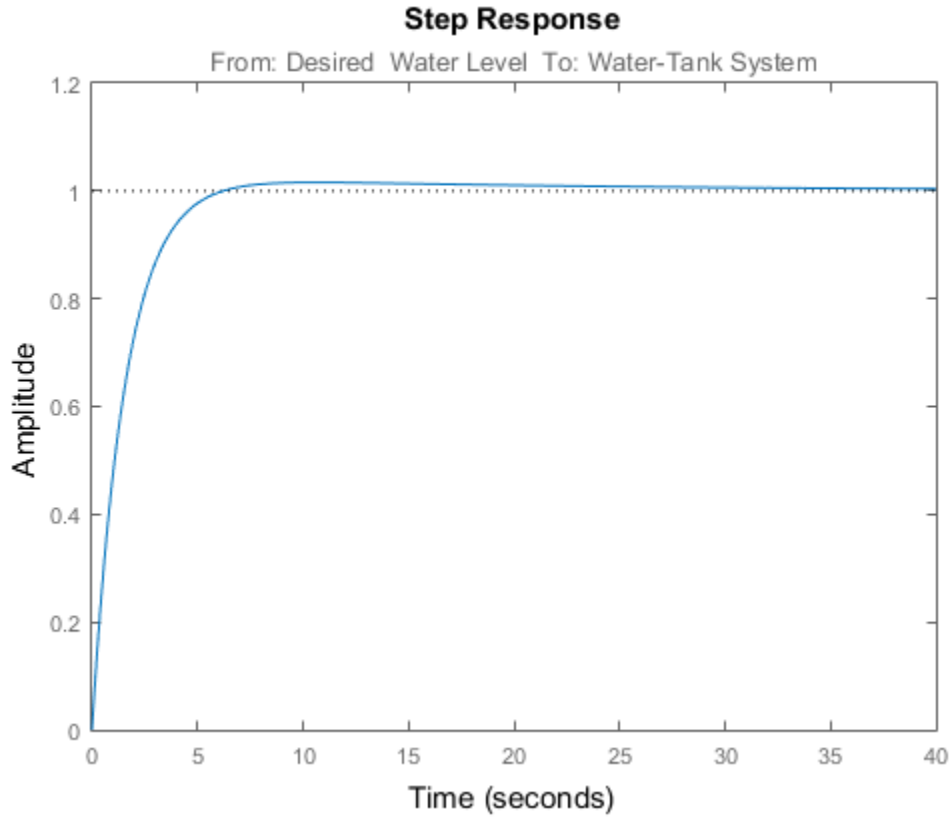
Use array indexing to extract from `linsys`the model obtained by linearizing the `watertank` model at `t = 3`, for `A = 10` and `b = 4`.

```
sg = linsys.SamplingGrid;
sys = linsys(:,:,sg.A == 10 & sg.B == 4 & sg.Time == 3);
```

The structure, `sg`, contains the sampling grid for all the models in `linsys`. The expression `sg.A == 10 & sg.B == 4 & sg.Time == 3` returns a logical array. Each entry of this array contains the logical evaluation of the expression for corresponding entries in `sg.A`, `sg.B`, and `sg.Time`. `sys`, a model array, contains all the `linsys` models that satisfy the expression.

View the step response for `sys`.

```
stepplot(sys)
```



See Also

Related Examples

- "Batch Linearize Model for Parameter Variations at Single Operating Point" on page 3-20
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41
- "Analyze Batch Linearization Results in Linear Analysis Tool" on page 3-55
- "Validate Batch Linearization Results" on page 3-90

Analyze Batch Linearization Results in Linear Analysis Tool

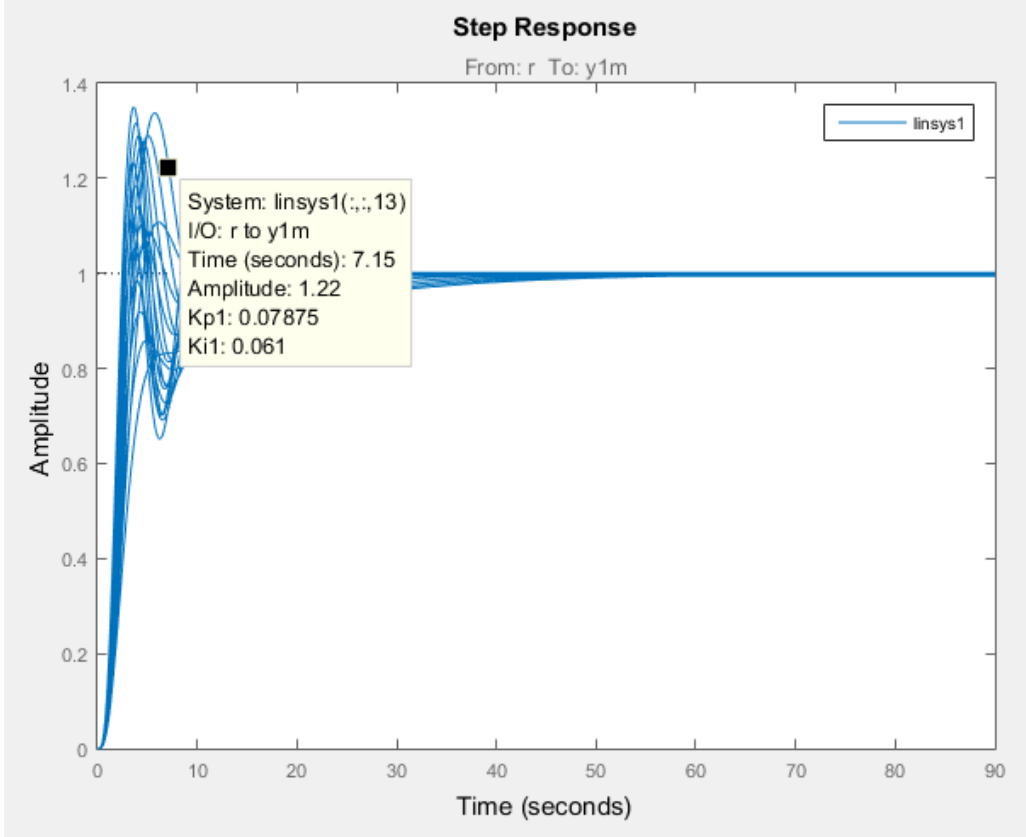
This example shows how to use response plots to analyze batch linearization results in Linear Analysis Tool. The term batch linearization results refers to linearizations for varying parameter values, such as illustrated in "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75. You can use the techniques illustrated in this example to analyze the frequency response, stability, and other system characteristics for batch linearization results.

View Parameters of a Response

For this example, suppose that you have batch linearized a model as described in "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75. You have generated a step response plot of an array of linear models computed for a 2-D parameter grid, with variations of outer-loop controller gains `Ki1` and `Kp1`.

When you perform batch linearization, Linear Analysis Tool generates a plot showing the responses of all linear models resulting from the linearization. You choose the response plot type, such as Step, Bode, or Nyquist, when you linearize. You can create additional plots at any time as described in "Analyze Results Using Linear Analysis Tool Response Plots" on page 2-146.

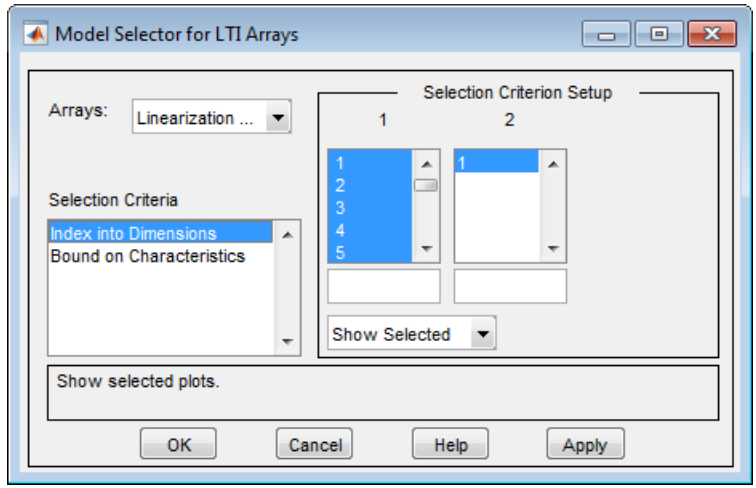
To view the parameters associated with a particular response, click the response on the plot.



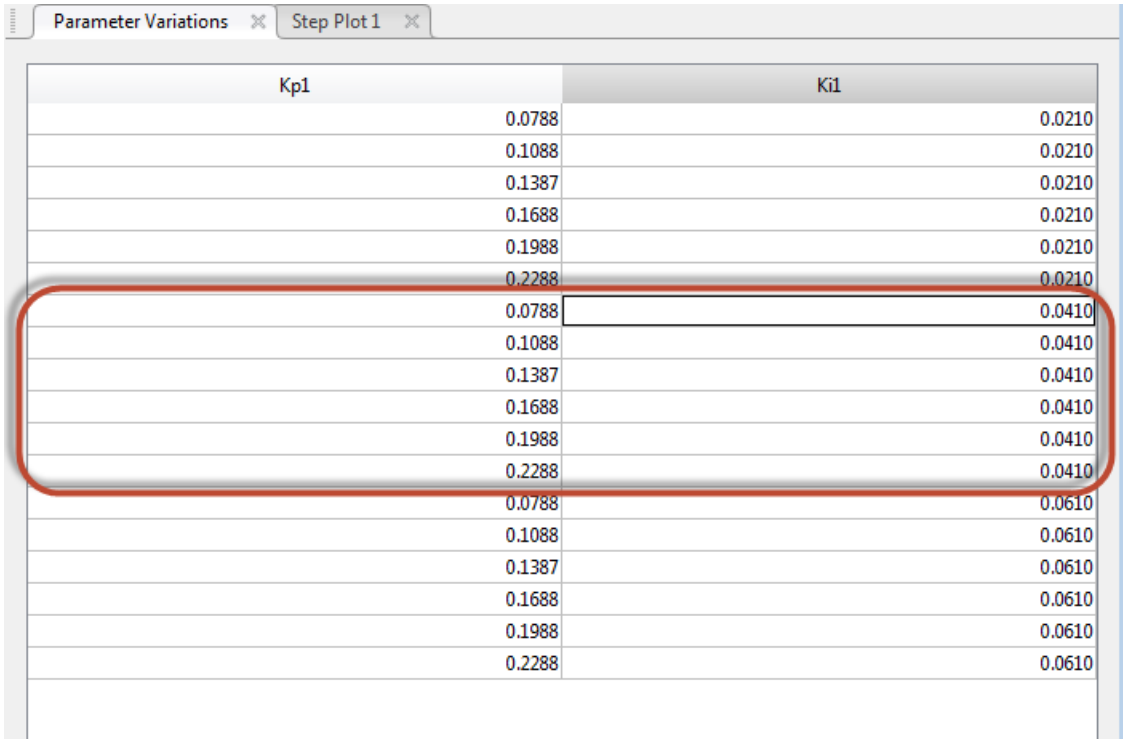
A data tip appears on the plot, providing information about the selected response and the related model. The last lines of the data tip show the parameter combination that yielded this response. For example, in this plot, the selected response corresponds to the model obtained by setting `Kp1` to `0.07875` and `Ki1` to `0.061`.

View Step Response of Subset of Results

Suppose you want to view the responses for only the models linearized at a specific `Ki1` value, the middle value `Ki1 = 0.0410`. Right-click the plot and select **Array Selector**. The Model Selector for LTI Arrays dialog box opens.



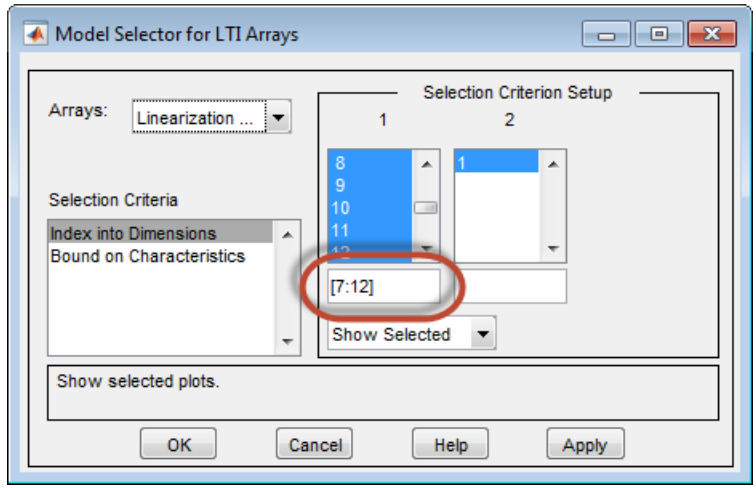
The **Selection Criterion Setup** panel contains two columns, one for each model array dimension of `linsys1`. Linear Analysis Tool flattens the 2-D parameter grid into a one-dimensional array, so that variations in both `Kp1` and `Ki1` are represented along the indices shown in column **1**. To determine which entries in this array correspond to `Ki1 = 0.0410`, examine the **Parameter Variations** table.



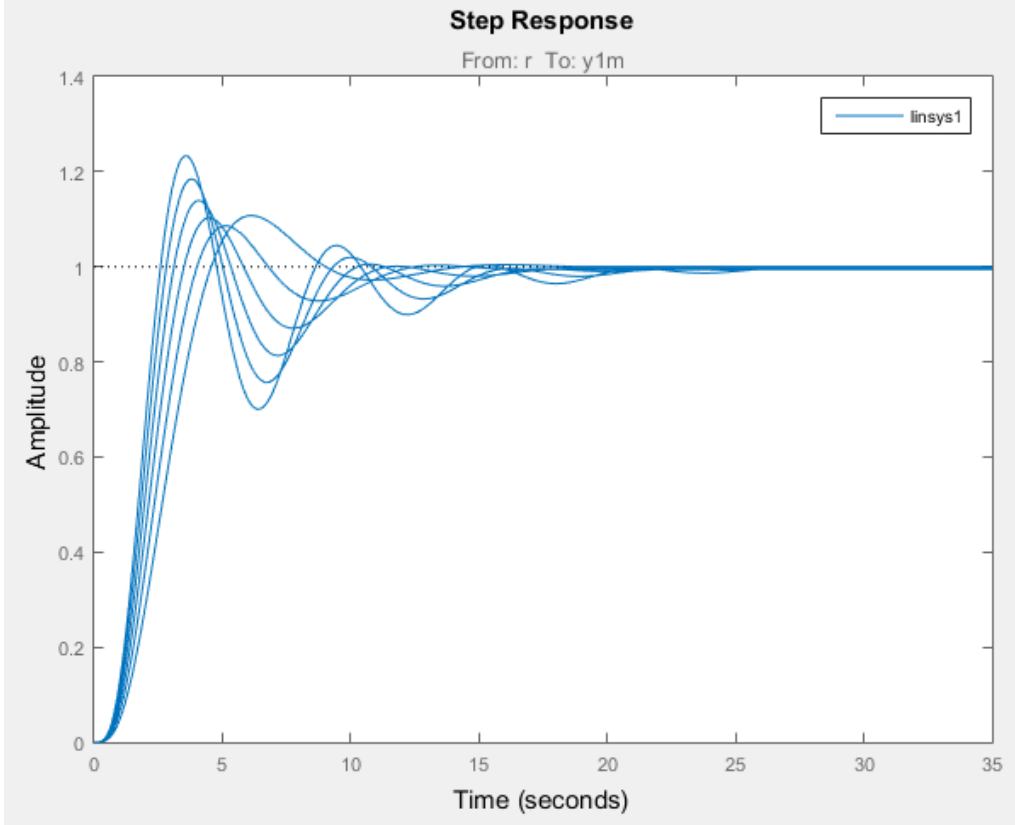
Kp1	Ki1
0.0788	0.0210
0.1088	0.0210
0.1387	0.0210
0.1688	0.0210
0.1988	0.0210
0.2288	0.0210
0.0788	0.0410
0.1088	0.0410
0.1387	0.0410
0.1688	0.0410
0.1988	0.0410
0.2288	0.0410
0.0788	0.0610
0.1088	0.0610
0.1387	0.0610
0.1688	0.0610
0.1988	0.0610
0.2288	0.0610

The `Ki1 = 0.0410` values are the seventh to twelfth entries in this table. Therefore, you want to select array indices 7–12.

In the Model Selector for LTI Arrays dialog box, enter `[7:12]` in the field below column **1**. The selection in the column changes to reflect this subset of the array.

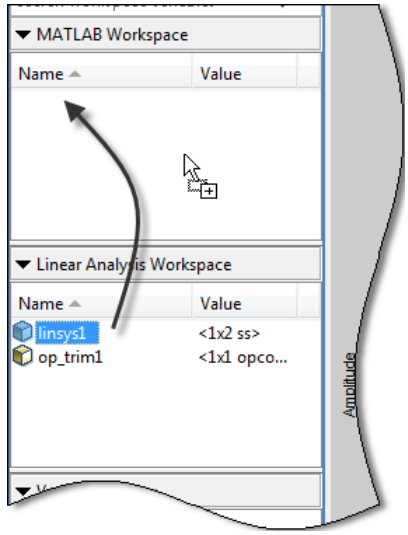


Click **OK**. The step plot displays responses only for the models with `Ki1 = 0.0410`.



Export Array to MATLAB Workspace

You can export the model array to the MATLAB workspace to perform further analysis or control design. To do so, in the Linear Analysis Tool, in the **Data Browser**, drag the array from Linear Analysis Workspace to the MATLAB workspace.



You can then use Control System Toolbox control design tools, such as the Linear System Analyzer (Control System Toolbox) app, to analyze linearization results. Or, use Control System Toolbox control design tools, such as `pidtune` or **Control System Designer**, to design controllers for the linearized systems.

See Also

More About

- "What Is Batch Linearization?" on page 3-2
- "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75

Specify Parameter Samples for Batch Linearization

In this section...

About Parameter Samples

Block parameters configure a Simulink model in several ways. For example, you can use block parameters to specify various coefficients or controller sample times. You can also use a discrete parameter, like the control input to a Multiport Switch block, to control the data path within a model. Varying the value of a parameter helps you understand its impact on the model behavior.

When using any of the Simulink Control Design linearization tools (or tuning with `slTuner` or Control System Tuner) you can specify a set of block parameter values at which to linearize the model. The full set of values is called a parameter grid or parameter samples. The tools batch-linearize the model, computing a linearization for each value in the parameter grid. You can vary multiple parameters, thus extending the parameter grid dimension. When using the command-line linearization tools, the `linearize` command or the `slLinearizer` or `slTuner` interfaces, you specify the parameter samples using a structure with fields `Name` and `Value`. In the Linear Analysis Tool or Control System Tuner, you use the graphical interface to specify parameter samples.

Which Parameters Can Be Sampled?

You can vary any model parameter whose value is given by a variable in the model workspace, the MATLAB workspace, or a data dictionary. In cases where the varying parameters are all tunable (Simulink), the linearization tools require only one model compilation to compute transfer functions for varying parameter values. This efficiency is especially advantageous for models that are expensive to compile repeatedly.

For more information, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10.

Vary Single Parameter at the Command Line

To vary the value of a single parameter for batch linearization with `linearize`, `slLinearizer`, or `slTuner`, specify the parameter grid as a structure having two fields. The `Name` field contains the name of the workspace variable that specifies the parameter. The `Value` field contains a vector of values for that parameter to take during linearization.

For example, the `Watertank` model has three parameters defined as MATLAB workspace variables, `a`, `b`, and `A`. The following commands specify a parameter grid for the single parameter for `A`.

```
param.Name = 'A';
param.Value = Avals;
```

Here, `Avals` is an array specifying the sample values for `A`.

The following table lists some common ways of specifying parameter samples.

Parameter Sample-Space Type	How to Specify the Parameter Samples
Linearly varying	`param.Value = linspace(A_min,A_max,num_samples)`
Logarithmically varying	`param.Value = logspace(A_min,A_max,num_samples)`
Random	`param.Value = rand(1,num_samples)`
Custom	`param.Value = custom_vector`

If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, suppose that `Kpid` is a vector of PID gains. The first entry in that vector, `Kpid`, is used as a gain value in a block in your model. Use the following commands to vary that gain using the values given in a vector `Kpvals`:

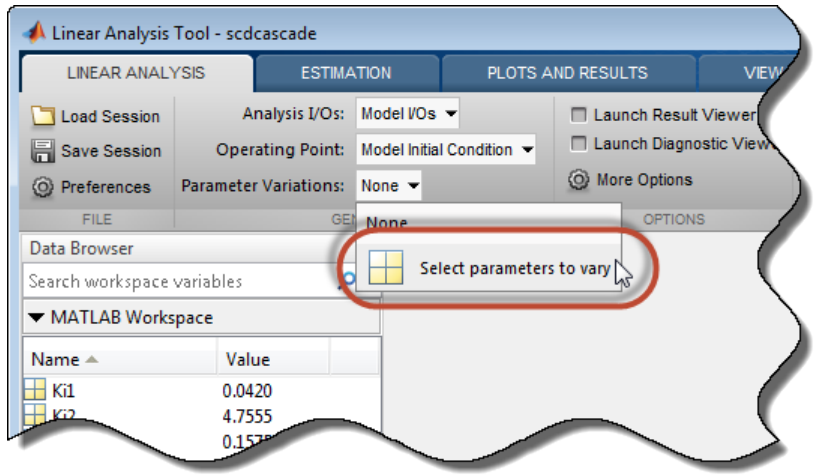
```
param.Name = 'Kpid(1)';
param.Value = Kpvals;
```

After you create the structure `param`:

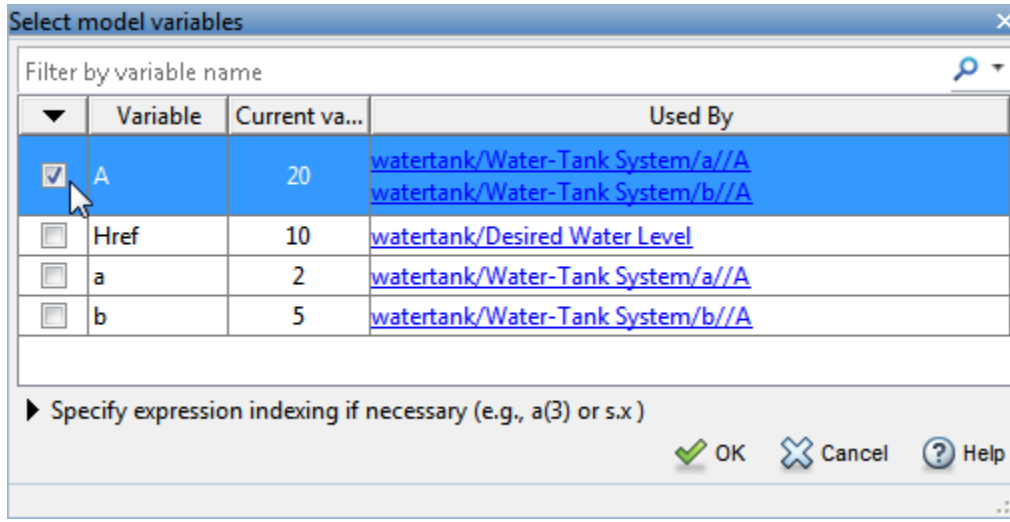
- Pass it to `linearize` as the `param` input argument.
- Pass it to `slLinearizer` as the `param` input argument, when creating an `slLinearizer` interface.
- Set the `Parameters` property of an existing `slLinearizer` interface to `param`.

Vary Single Parameter in Graphical Tools

To specify variations of a single parameter for batch linearization in Linear Analysis Tool, in the **Linear Analysis** tab, in the **Parameter Variations** drop-down list, click `Select parameters to vary`. (In **Control System Tuner**, the **Parameter Variations** drop-down list is in the **Control System** tab.)



Click  **Manage Parameters**. In the Select model variables dialog box, check the parameter to vary. The table lists all variables in the MATLAB workspace and the model workspace that are used in the model, whether tunable or not.

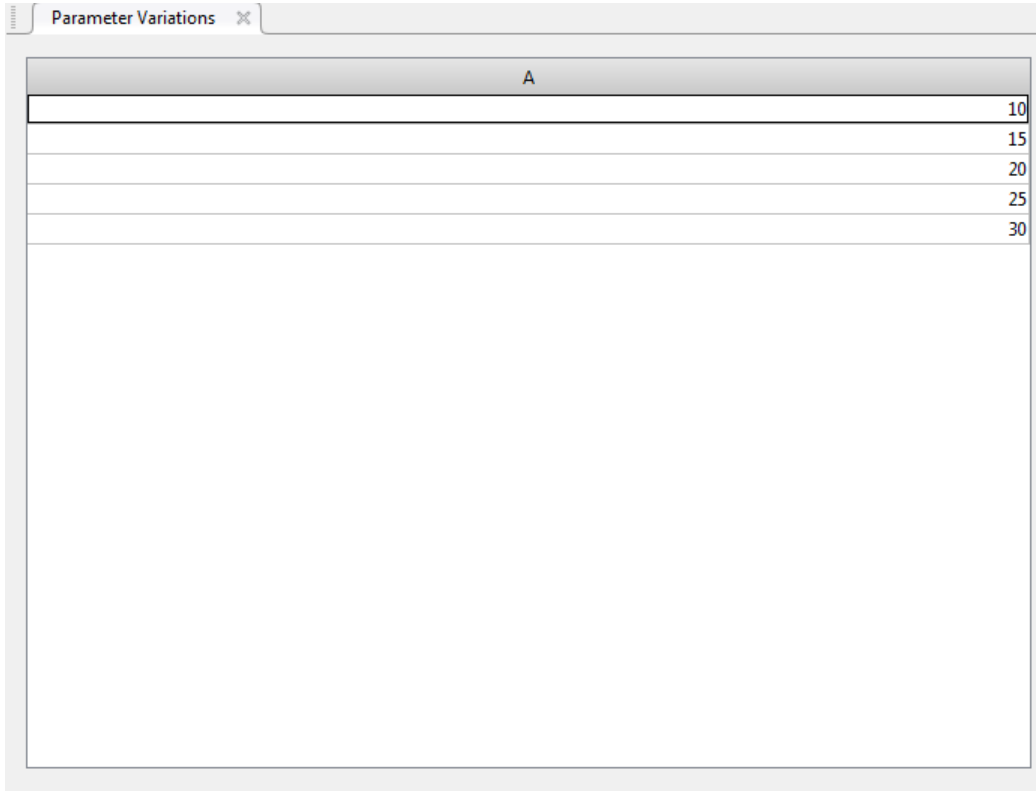


Note If the parameter is not a scalar variable, click **Specify expression indexing if necessary** and enter an expression that resolves to a numeric scalar value. For example, if `A` is a vector, enter `A(3)` to specify the third entry in `A`. If `A` is a structure and the scalar parameter you want to vary is the `Value` field of that structure, enter `A.Value`. The indexed variable appears in the variable list.

Click **OK**. The selected variable appears in the **Parameter Variations** table. Use the table to specify parameter values manually, or generate values automatically.

Manually Specify Parameter Values

To specify the values manually, add rows to the table by clicking **Insert Row** and selecting either `Insert Row Above` or `Insert Row Below`. Then, edit the values in the table as needed.



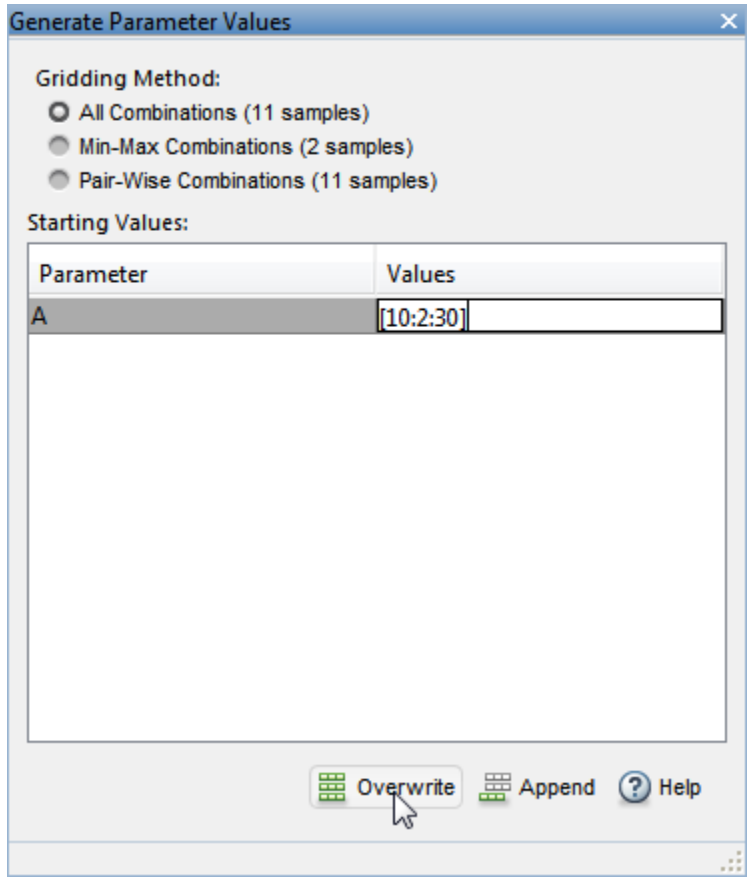
When you return to the **Linear Analysis** tab and linearize the model, Linear Analysis Tool linearizes at all parameter values listed in the **Parameter Variations** table.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Automatically Generate Parameter Values

To generate values automatically, click **Generate Values**. In the Generate Parameter Values dialog box, in the **Values** column, enter an expression for the

parameter values you want for the variable. For example, enter an expression such as `linspace(A_min,A_max,num_samples)`, or `[10:2:30]`.



Click **Overwrite** to replace the values in the **Parameter Variations** table with the generated values.

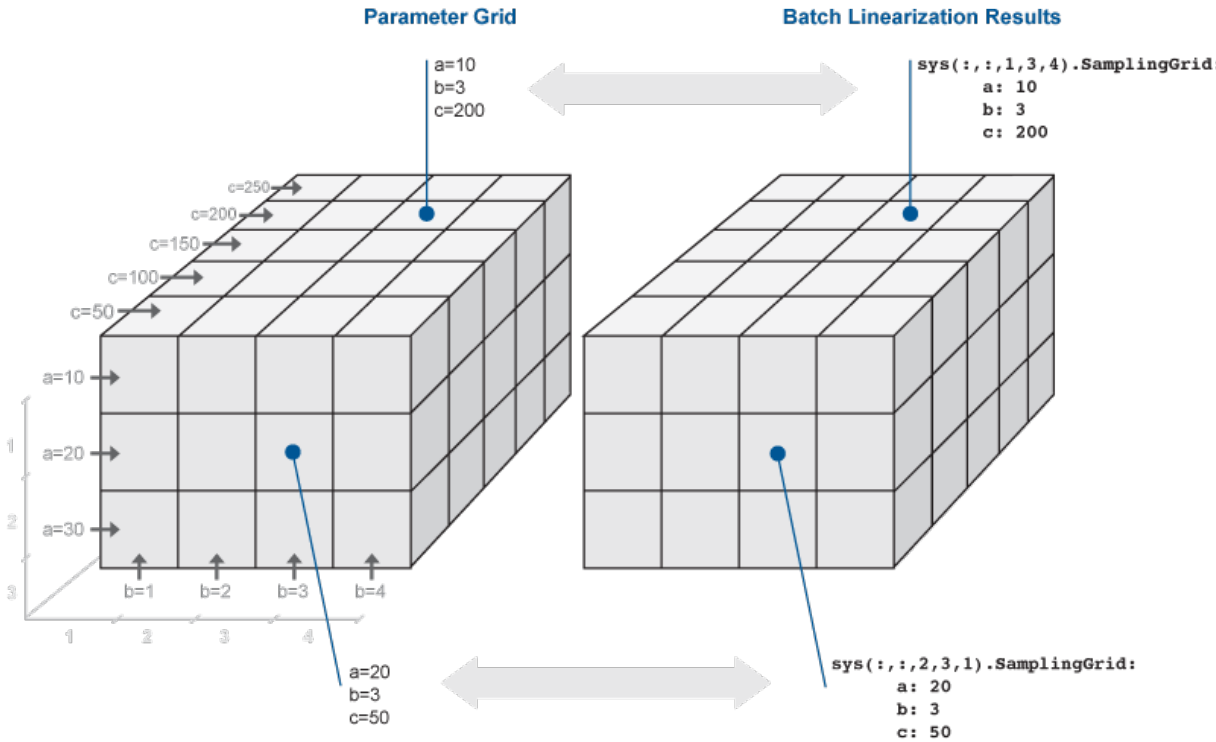
When you return to the **Linear Analysis** tab and linearize the model, Linear Analysis Tool computes a linearization for each of these parameter values.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the

Parameter Variations tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Multi-Dimension Parameter Grids

When you vary more than one parameter at a time, you generate parameter grids of higher dimension. For example, varying two parameters yields a parameter matrix, and varying three parameters yields a 3-D parameter grid. Consider the following parameter grid:



Here, you vary the values of three parameters, *a*, *b*, and *c*. The samples form a 3-by-4-by-5 grid. When batch linearizing your model, the `ss` model array, `sys`, is the batch result. Similarly, when batch trimming your model, you get an array of operating point objects.

Vary Multiple Parameters at the Command Line

To vary the value of multiple parameters for batch linearization with `linearize`, `slLinearizer`, or `slTuner`, specify parameter samples as a structure array. The structure has an entry for each parameter whose value you vary. The structure for each parameter is the same as described in "Vary Single Parameter at the Command Line" on page 3-63. You can specify the `Value` field for a parameter to be an array of any dimension. However, the size of the `Value` field must match for all parameters. Corresponding array entries for all the parameters, also referred to as a parameter grid point, must map to a desired parameter combination. When the software linearizes the model, it computes a linearization—an `ss` model—for each grid point. The software populates the `SamplingGrid` property of each linearized model with information about the parameter grid point that the model corresponds to.

Specify Full Grid

Suppose that your model has two parameters whose values you want to vary, a and b:

$a = \{a1, a2\}$

$b = \{b1, b2\}$

You want to linearize the model for every combination of a and b, also referred to as a full grid:

$$\begin{Bmatrix} (a_1, b_1), & (a_1, b_2) \\ (a_2, b_1), & (a_2, b_2) \end{Bmatrix}$$

Create a rectangular parameter grid using `ndgrid`.

```
a1 = 1;
a2 = 2;
a = [a1 a2];

b1 = 3;
b2 = 4;
b = [b1 b2];

[A,B] = ndgrid(a,b)
```

```
>> A

A =
```

```
     1     1
     2     2

>> B

B =

     3     4
     3     4
```

Create the structure array, `params`, that specifies the parameter grid.

```
params(1).Name = 'a';
params(1).Value = A;

params(2).Name = 'b';
params(2).Value = B;
```

In general, to specify a full grid for *N* parameters, use `ndgrid` to obtain *N* grid arrays.

```
[P1,...,PN] = ndgrid(p1,...,pN);
```

Here, p_1,...,p_N are the parameter sample vectors.

Create a 1 x *N* structure array.

```
params(1).Name = 'p1';
params(1).Value = P1;
...
params(N).Name = 'pN';
params(N).Value = PN;
```

Specify Subset of Full Grid

If your model is complex or you vary the value of many parameters, linearizing the model for the full grid can become expensive. In this case, you can specify a subset of the full grid using a table-like approach. Using the example in "Specify Full Grid" on page 3-69, suppose you want to linearize the model for the following combinations of *a* and *b*:
$\{(a_1,b_1),(a_1,b_2)\}$

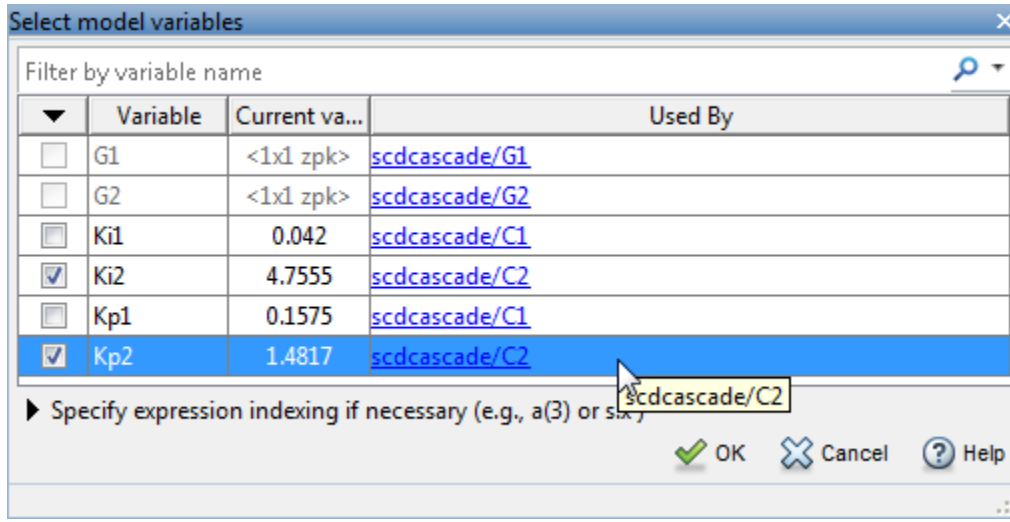
Create the structure array, `params`, that specifies this parameter grid.

```
A = [a1 a1];
params(1).Name = 'a';
```

```
params(1).Value = A;

B = [b1 b2];
params(2).Name = 'b';
params(2).Value = B;
```

Vary Multiple Parameters in Graphical Tools

To vary the value of multiple parameters for batch linearization in Linear Analysis Tool or Control System Tuner, open the Select model variables dialog box, as described in "Vary Single Parameter in Graphical Tools" on page 3-64. In the dialog box, check all variables you want to vary.



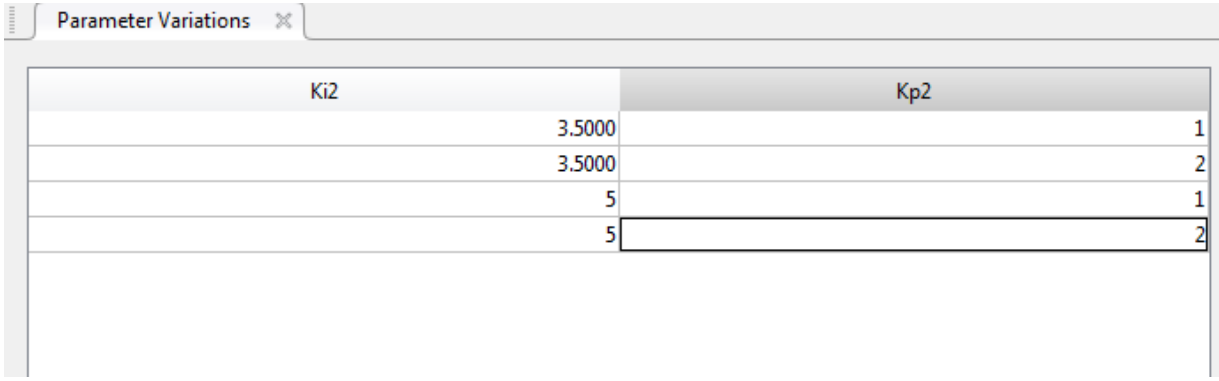
Note If a parameter you want to vary is not a scalar variable, click **Specify expression indexing if necessary** and enter an expression that resolves to a scalar value. For example, if `A` is a vector, enter `A(3)` to specify the third entry in `A`. If `A` is a structure and the scalar parameter you want to vary is the `Value` field of that structure, enter `A.Value`. The indexed variable appears in the variable list.

Click **OK**. The selected variables appear in the **Parameter Variations** table. Each column in the table corresponds to one selected variable. Each row in the table

represents one full set of parameter values at which to linearize the model. When you linearize, Linear Analysis Tool computes as many linear models as there are rows in the table. Use the table to specify combinations of parameter values manually, or generate value combinations automatically.

Manually Specify Parameter Values

To specify the values manually, add rows to the table by clicking **Insert Row** and selecting either `Insert Row Above` or `Insert Row Below`. Then, edit the values in the table as needed. For example, the following table specifies linearization at four parameter-value pairs: `(Ki2,Kp2)` = (3.5,1), (3.5,2), (5,1), and (5,2).



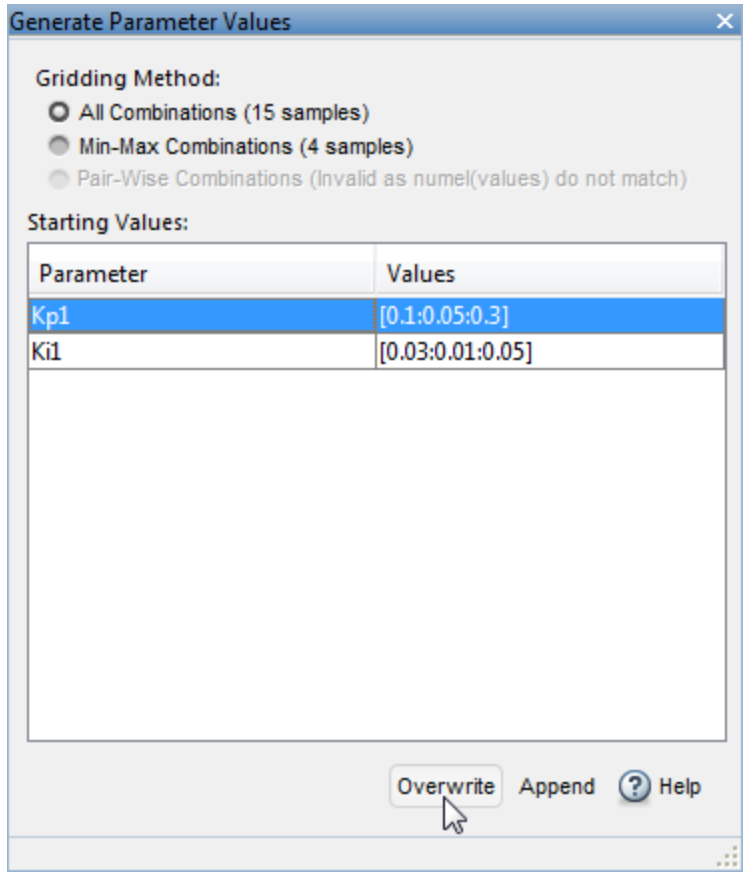
When you return to the **Linear Analysis** tab and linearize the model, Linear Analysis Tool computes a linearization for each of these parameter-value pairs.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Automatically Generate Parameter Values

To generate values automatically, click **Generate Values**. In the Generate Parameter Values dialog box, in the Values column, enter an expression for the parameter values you want for each variable, such as

`linspace(A_min,A_max,num_samples)`, or `[10:2:30]`. For example, the following entry generates parameter-value pairs for all possible combinations of `Kp1 = [0.1,0.15,0.2,0.25,0.3]` and `Kp2 = [0.03,0.04,0.05]`.



Click **Overwrite** to replace the values in the **Parameter Variations** table with the generated values.

When you return to the **Linear Analysis** tab and linearize the model, Linear Analysis Tool computes a linearization for each of these parameter-value pairs.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the

Parameter Variations tab, click  **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

See Also

`linearize` | `linspace` | `logspace` | `ndgrid` | `rand` | `slLinearizer` | `slTuner`

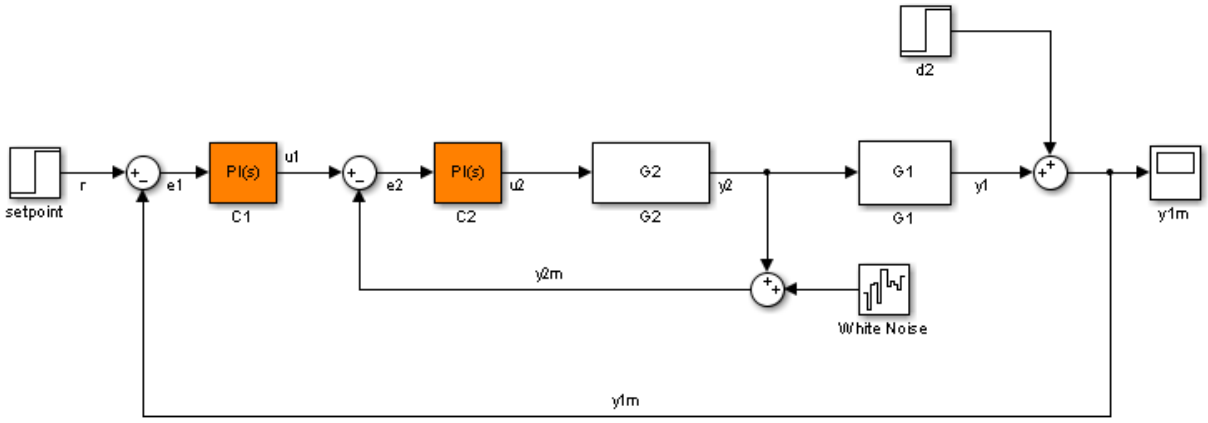
More About

- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10
- “Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool” on page 3-75
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32

Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool

This example shows how to use the Linear Analysis Tool to batch linearize a Simulink model. You vary model parameter values and obtain multiple open-loop and closed-loop transfer functions from the model.

The `scdcascade` model used for this example contains a pair of cascaded feedback control loops. Each loop includes a PI controller. The plant models, G1 (outer loop) and G2 (inner loop), are LTI models. In this example, you use Linear Analysis Tool to vary the PI controller parameters and analyze the inner-loop and outer-loop dynamics.

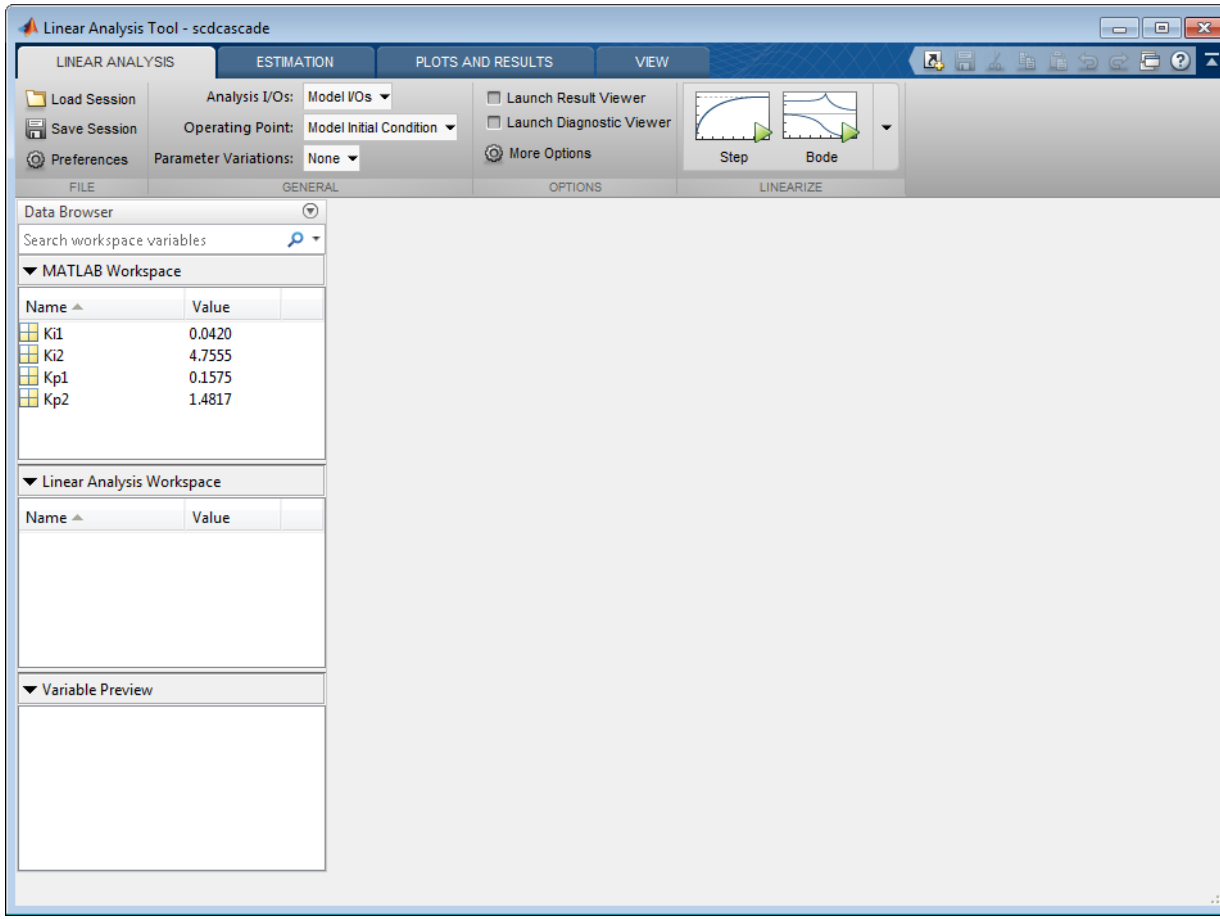


Open Linear Analysis Tool for the Model

At the MATLAB command line, open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```

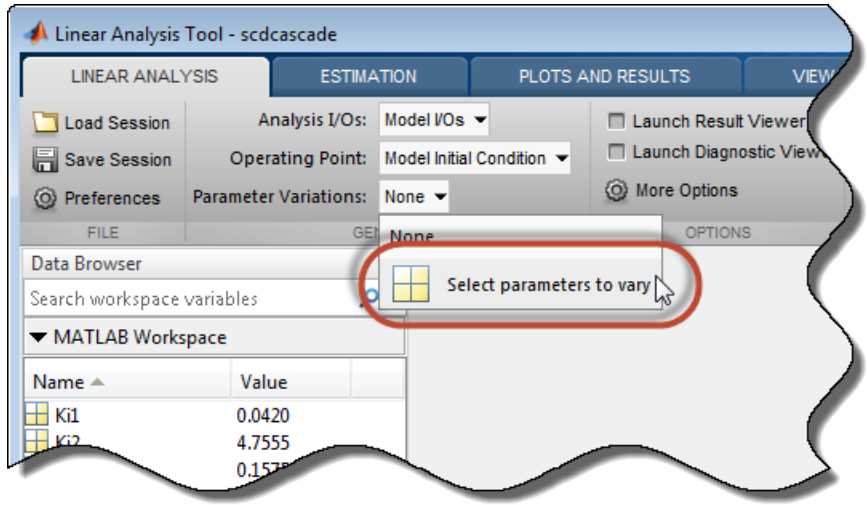
In the model window, select **Analysis > Control Design > Linear Analysis** to open the Linear Analysis Tool for the model.



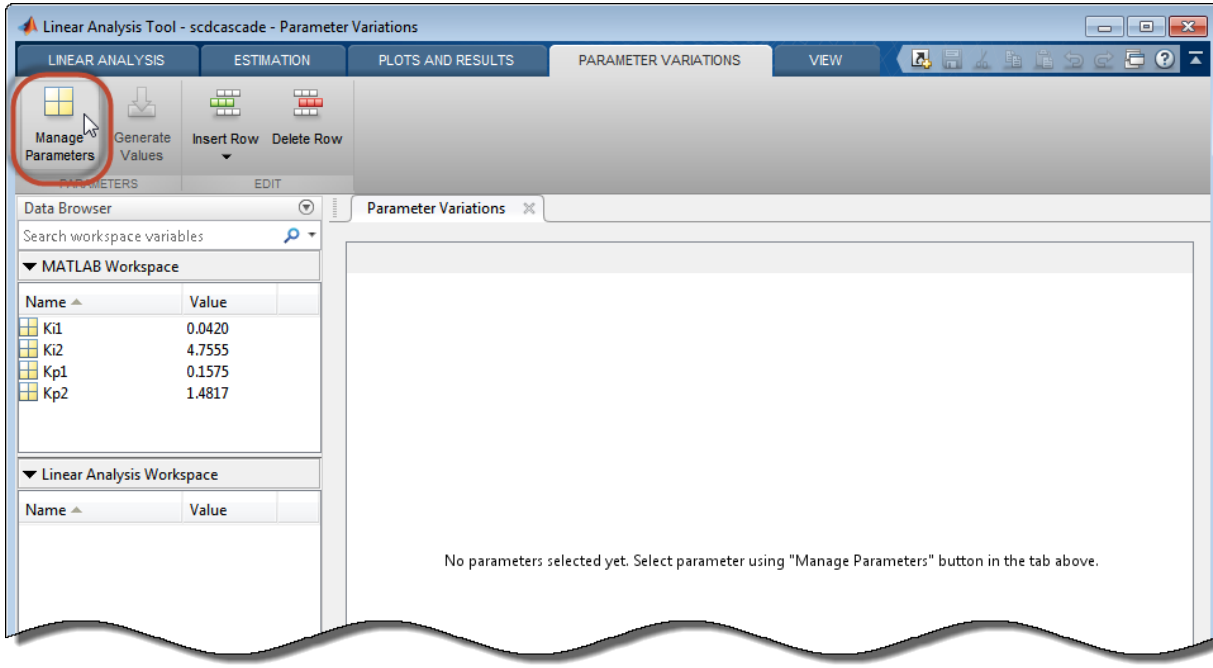
Vary the Inner-Loop Controller Gains

To analyze the behavior of the inner loop, very the gains of the inner-loop PI controller, `C2`. As you can see by inspecting the controller block, the proportional gain is the variable `Kp2`, and the integral gain is `Ki2`. Examine the performance of the inner loop for two different values of each of these gains.

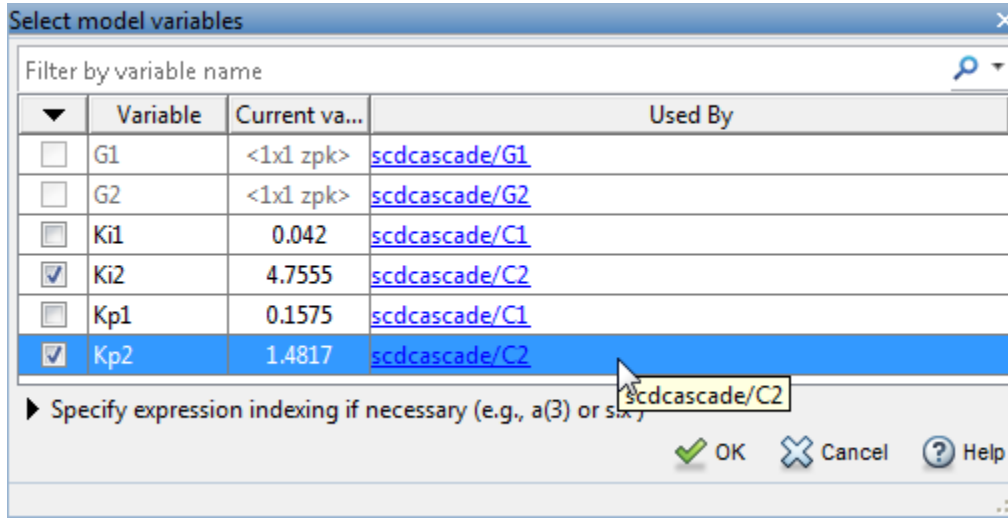
In the **Parameter Variations** drop-down list, click  `Select parameters to vary`.



The **Parameter Variations** tab opens. click **Manage Parameters**.

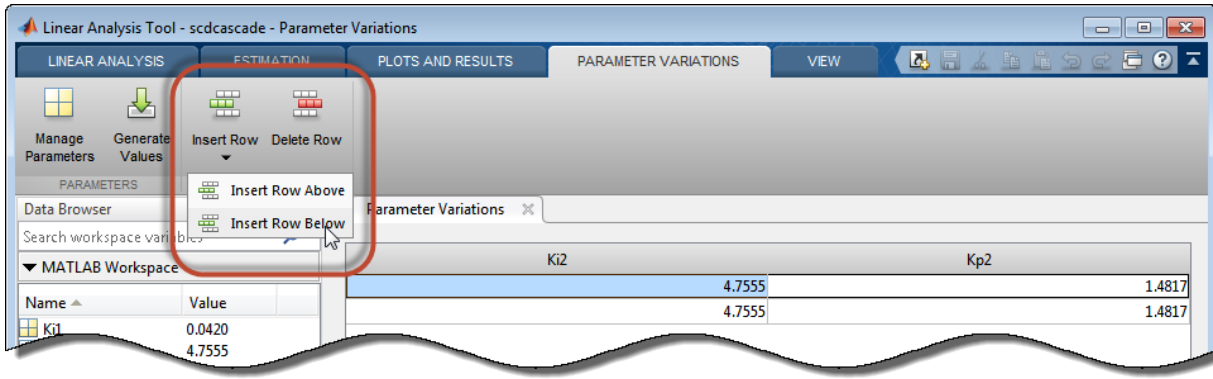


In the Select model variables dialog box, check the parameters to vary, `Ki2` and `Kp2`.

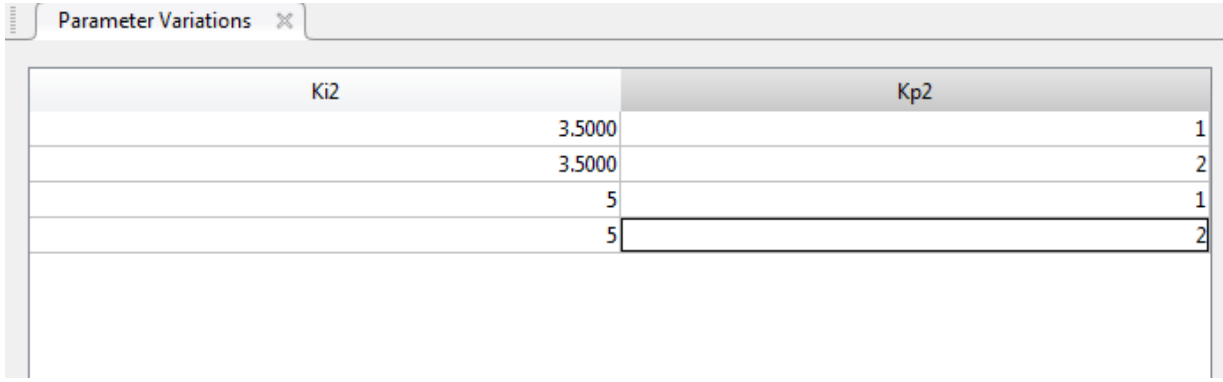


The selected variables appear in the **Parameter Variations** table. Each column in the table corresponds to one of the selected variables. Each row in the table represents one `(Ki2,Kp2)` pair at which to linearize. These parameter-value combinations are called parameter samples. When you linearize, Linear Analysis Tool computes as many linear models as there are parameter samples, or rows in the table.

Specify the parameter samples at which to linearize the model. For this example, specify four `(Ki2,Kp2)` pairs, `(Ki2,Kp2)` = (3.5,1), (3.5,2), (5,1), and (5,2). Enter these values in the table manually. To do so, select a row in the table. Then, select **Insert Row** > **Insert Row Below** twice.



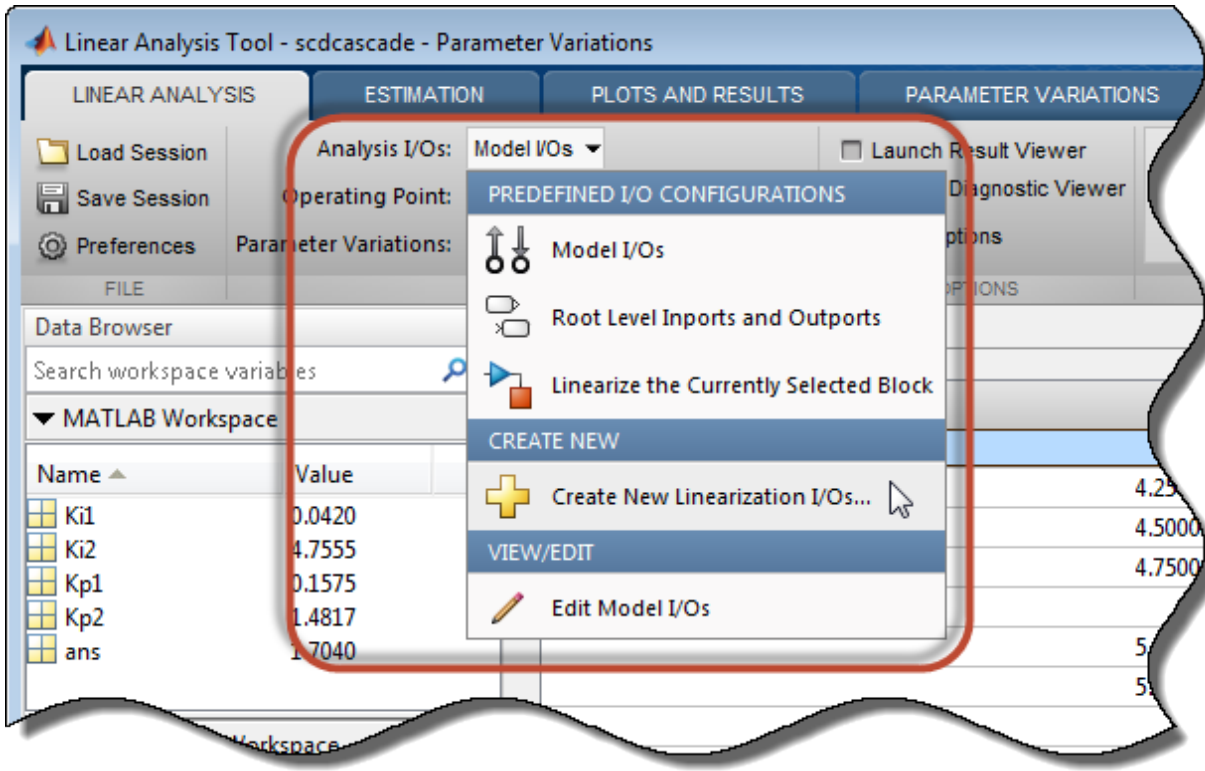
Edit the values in the table as shown to specify the four `(Ki2,Kp2)` pairs.



Tip For more details about specifying parameter values, see "Specify Parameter Samples for Batch Linearization" on page 3-62

Analyze the Inner Loop Closed-Loop Response

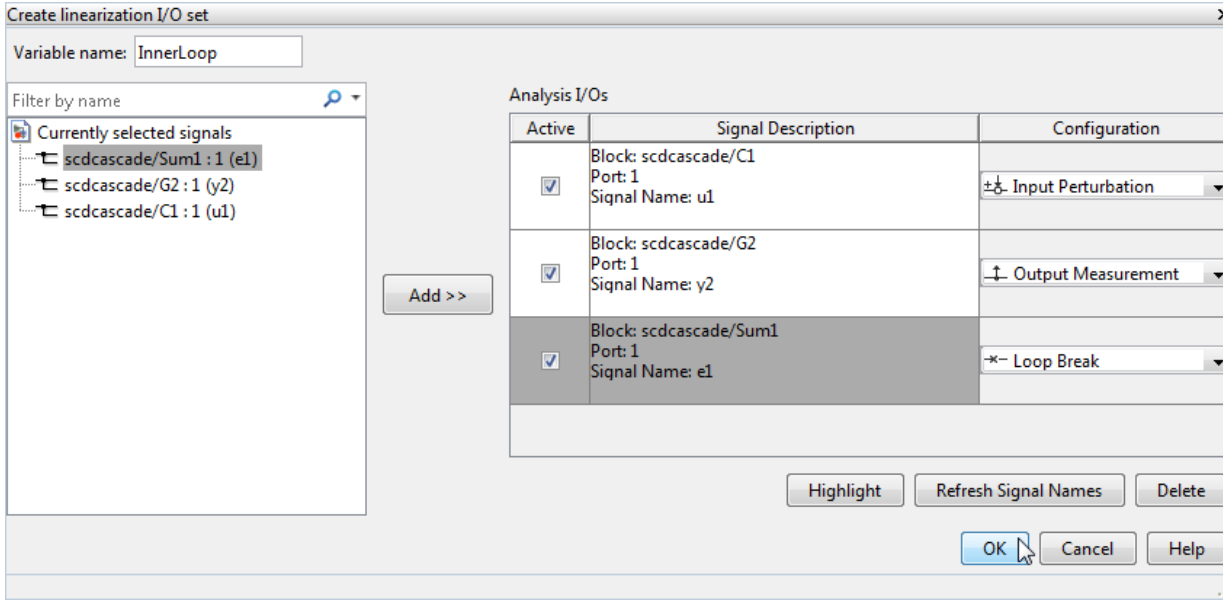
To analyze the inner-loop performance, extract a transfer function from the inner-loop input `u1` to the inner-plant output `y2`, computed with the outer loop open. To specify this I/O for linearization, in the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Create New Linearization I/Os`.



Specify the I/O set by creating:

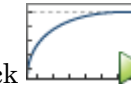
- An input perturbation point at `u1`
- An output measurement point at `y2`
- A loop break at `e1`

Name the I/O set by typing `InnerLoop` in the **Variable name** field of the Create linearization I/O set dialog box. The configuration of the dialog box is as shown.

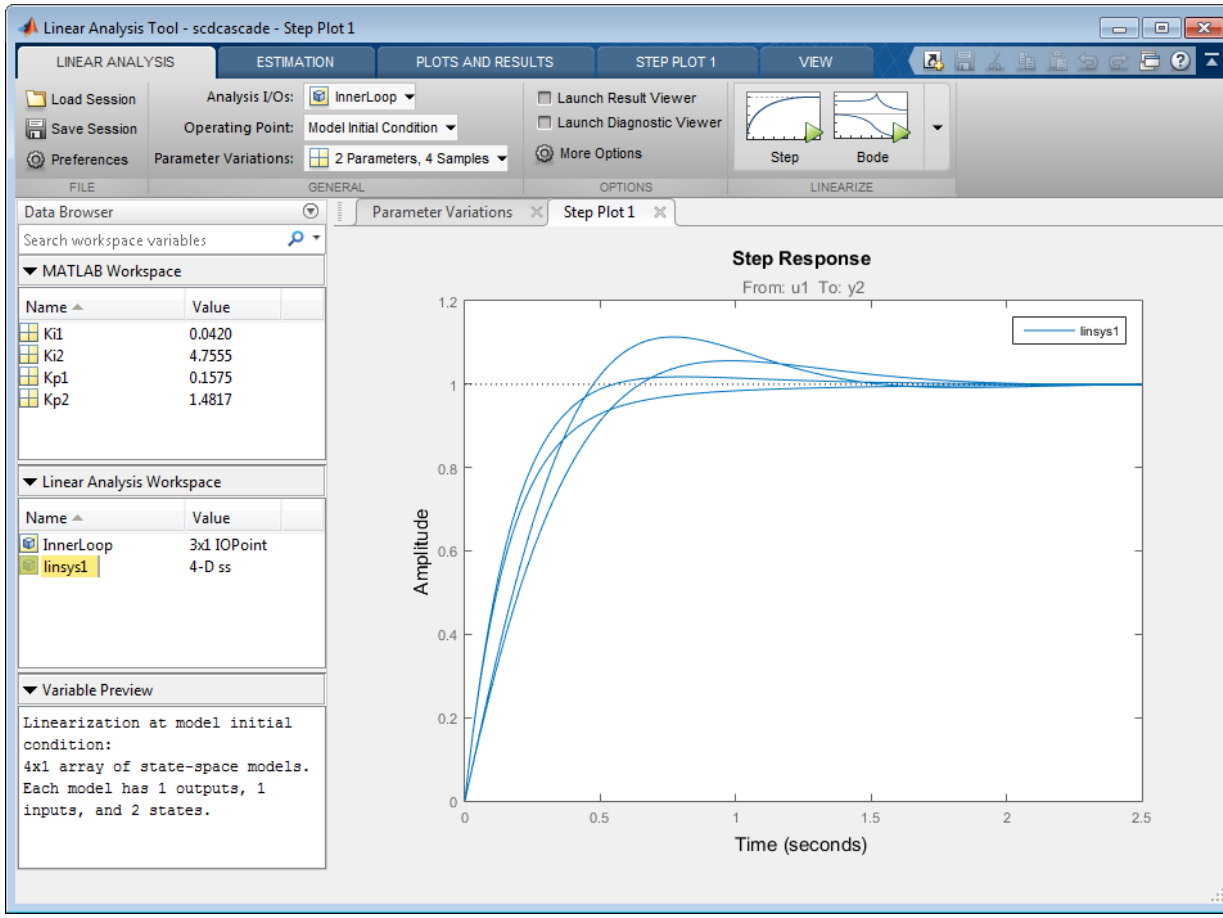


Tip For more information about specifying linearization I/Os, see "Specify Portion of Model to Linearize" on page 2-13.

Click **OK**.

Now that you have specified the parameter variations and the analysis I/O set for the inner loop, linearize the model and examine a step response plot. Click  **Step**.

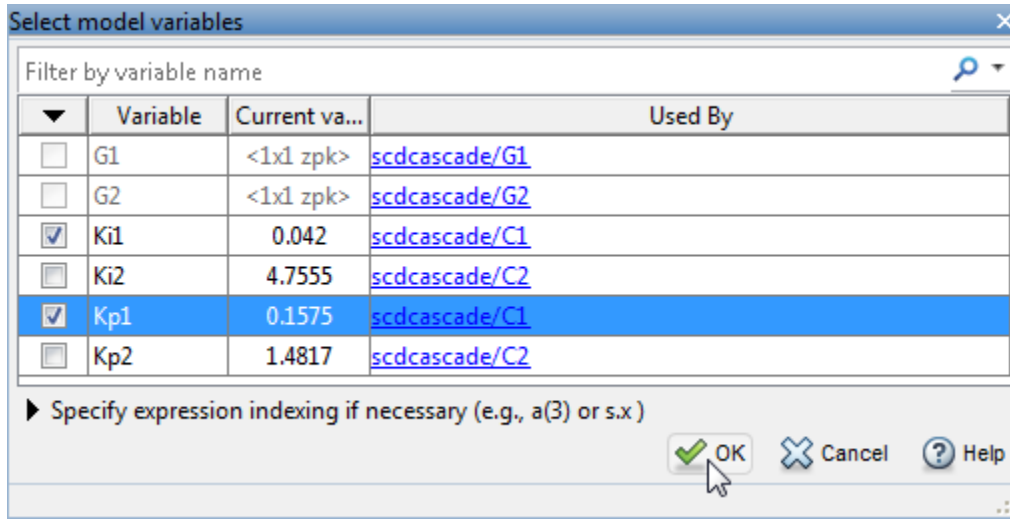
Linear Analysis Tool linearizes the model at each of the parameter samples you specified in the Parameter Variations table. A new variable, `linsys1`, appears in the Linear Analysis Workspace section of the Data Browser. This variable is an array of state-space (`ss`) models, one for each `(Ki2,Kp2)` pair. The plot shows the step responses of all the entries in `linsys1`. This plot gives you a sense of the range of step responses of the system in the operating ranges covered by the parameter grid.



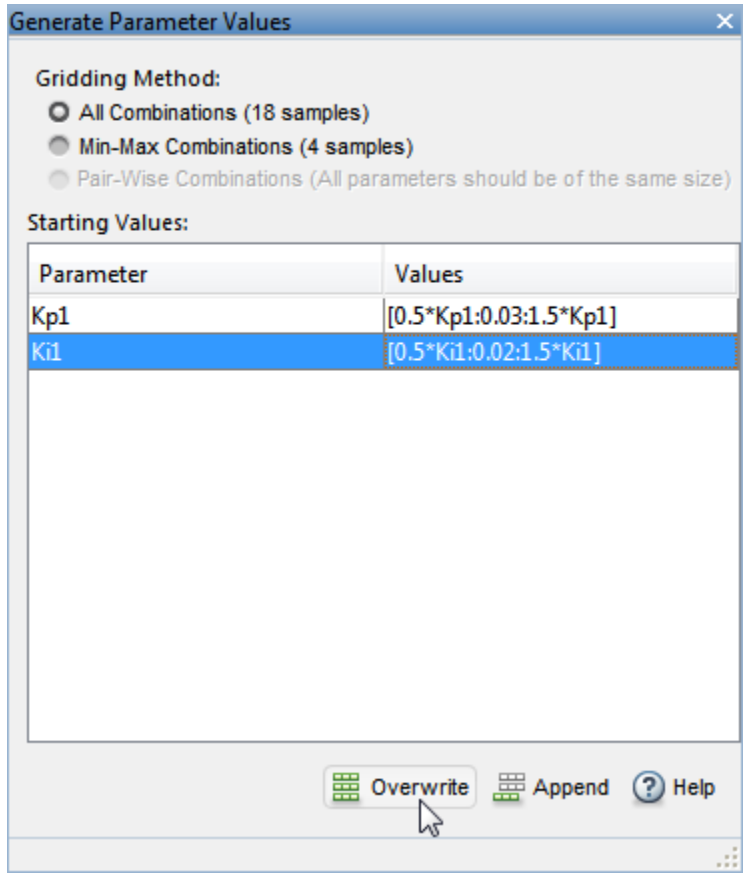
Vary the Outer-Loop Controller Gains

Examine the overall performance of the cascaded control system for varying values of the outer-loop controller, `C1`. To do so, vary the coefficients `Ki1` and `Kp1`, while keeping `Ki2` and `Kp2` fixed at the values specified in the model.

In the **Parameter Variations** tab, click  **Manage Parameters**. Clear the `Ki2` and `Kp2` checkboxes, and check `Ki1` and `Kp1`. Click **OK**.



Use Linear Analysis Tool to generate parameter values automatically. Click  **Generate Values**. In the **Values** column of the Generate Parameter Values table, enter an expression specifying the possible values for each parameter. For example, vary `Kp1` and `Ki1` by ± 50% of their nominal values, by entering expressions as shown.

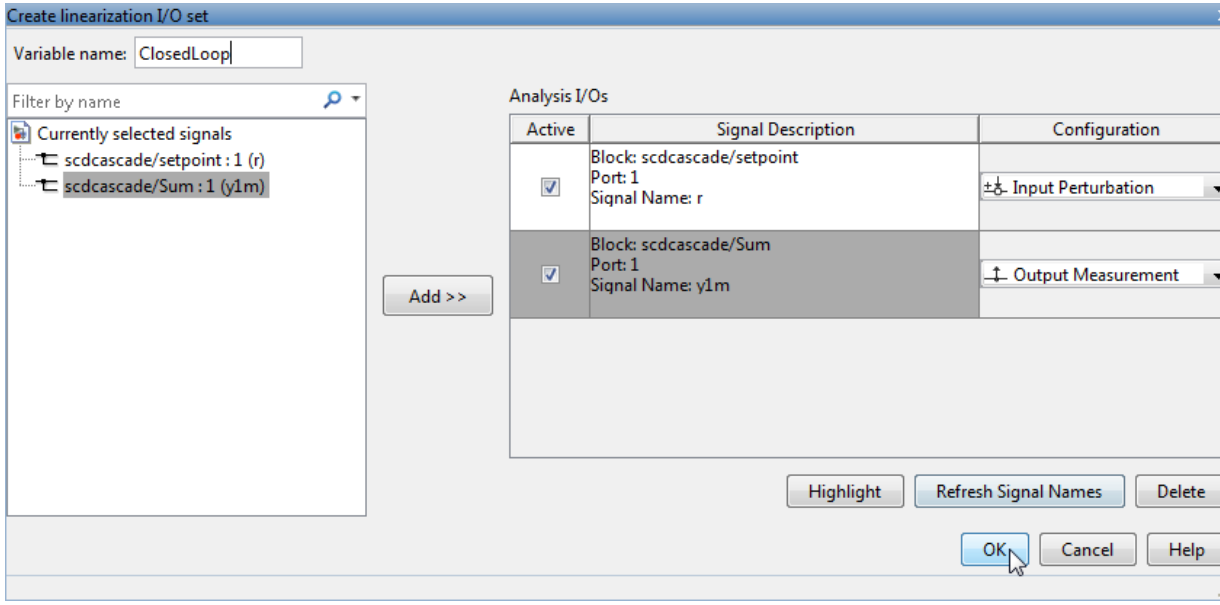


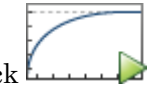
The **All Combinations** gridding method generates a complete parameter grid of `(Kp1,Ki1)` pairs, to compute a linearization at all possible combinations of the specified values. Click **Overwrite** to replace all values in the Parameter Variations table with the generated values.

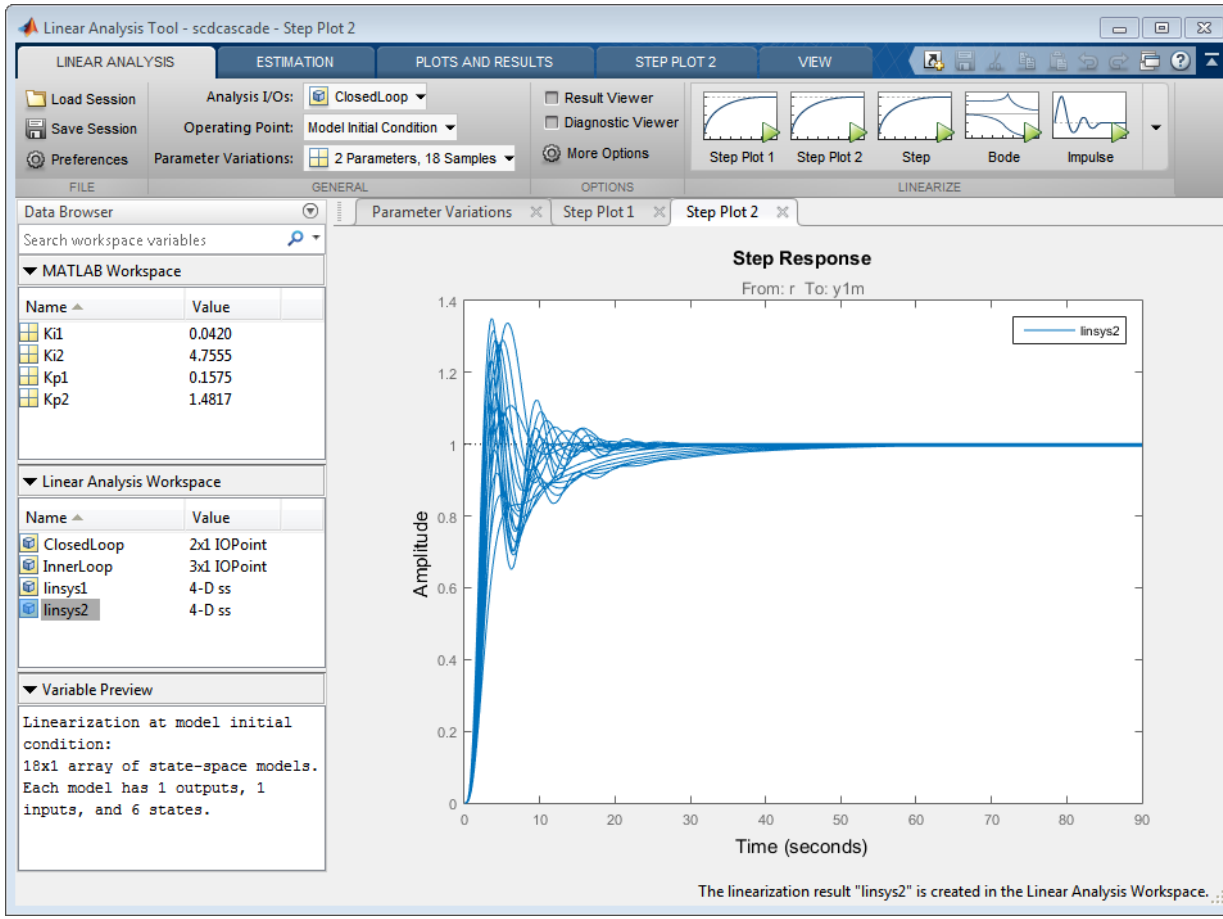
Parameter Variations | Step Plot 1

Kp1	Ki1
0.0788	0.0210
0.1088	0.0210
0.1387	0.0210
0.1688	0.0210
0.1988	0.0210
0.2288	0.0210
0.0788	0.0410
0.1088	0.0410
0.1387	0.0410
0.1688	0.0410
0.1988	0.0410
0.2288	0.0410
0.0788	0.0610
0.1088	0.0610
0.1387	0.0610
0.1688	0.0610
0.1988	0.0610
0.2288	0.0610

Because you want to examine the overall closed-loop transfer function of the system, create a new linearization I/O set. In the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Create New Linearization I/Os`. Configure `r` as an input perturbation point, and the system output `y1m` as an output measurement. Click **OK**.



Linearize the model with the parameter variations and examine the step response of the resulting models. Click  **Step** to linearize and generate a new plot for the new model array, `linsys2`.



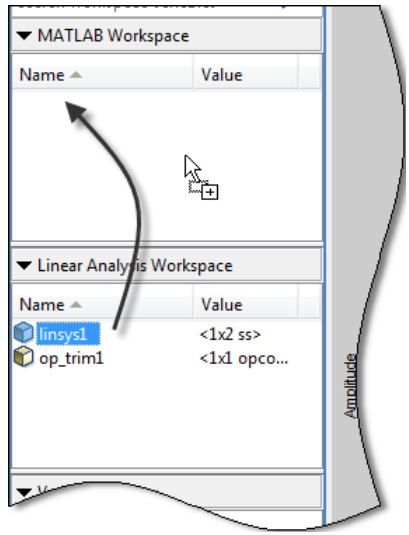
The step plot shows the responses of every model in the array. This plot gives you a sense of the range of step responses of the system in the operating ranges covered by the parameter grid.

Note Although the new plot reflects the new set of parameter variations, `Step Plot 1` and `linsys1` are unchanged. That plot and array still reflect the linearizations obtained with the inner-loop parameter variations.

Further Analysis of Batch Linearization Results

The results of both batch linearizations, `linsys1` and `linsys2`, are arrays of state-space (`ss`) models. Use these arrays for further analysis in any of several ways:

- Create additional analysis plots, such as Bode plots or impulse response plots, as described in “Analyze Results Using Linear Analysis Tool Response Plots” on page 2-146.
- Examine individual responses in analysis plots as described in “Analyze Batch Linearization Results in Linear Analysis Tool” on page 3-55.
- Drag the array from Linear Analysis Workspace to the MATLAB workspace.



You can then use Control System Toolbox control design tools, such as the Linear System Analyzer (Control System Toolbox) app, to analyze linearization results. Or, use Control System Toolbox control design tools, such as `pidtune` or **Control System Designer**, to design controllers for the linearized systems.

Also see “Validate Batch Linearization Results” on page 3-90 for information about validating linearization results in the MATLAB workspace.

See Also

More About

- “Validate Batch Linearization Results” on page 3-90
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10
- “Specify Parameter Samples for Batch Linearization” on page 3-62
- “Analyze Batch Linearization Results in Linear Analysis Tool” on page 3-55
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20

Validate Batch Linearization Results

When you batch linearize a model, the software returns a model array containing the linearized models. There are two ways to validate a linearized model, but both methods have some computational overhead. This overhead can make validating each model in the batch linearization results infeasible. Therefore, it can be cost effective to validate either a single model or a subset of the batch linearization results. You can use linear analysis plots and commands to determine the validation candidates. For information regarding the tools that you can use for such analysis, see "Linear Analysis" (Control System Toolbox).

You can validate a linearization using the following approaches:

- Obtain a frequency response estimation of the nonlinear model, and compare its response to that of the linearized model. For an example, see "Validate Linearization In Frequency Domain" on page 2-140.
- Simulate the nonlinear model and compare its time-domain response to that of the linearized model. For an example, see "Validate Linearization In Time Domain" on page 2-136.

See Also

`linearize` | `slLinearizer`

Related Examples

- "Analyze Batch Linearization Results in Linear Analysis Tool" on page 3-55
- "Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48

Approximating Nonlinear Behavior Using an Array of LTI Systems

This example shows how to approximate the nonlinear behavior of a system as an array of interconnected LTI models.

The example describes linear approximation of pitch axis dynamics of an airframe over a range of operating conditions. The array of linear systems thus obtained is used to create a Linear Parameter Varying (LPV) representation of the dynamics. The LPV model serves as an approximation of the nonlinear pitch dynamics.

About Linear Parameter Varying (LPV) Models

In many situations the nonlinear dynamics of a system need to be approximated using simpler linear systems. A single linear system provides a reasonable model for behavior limited to a small neighborhood around an operating point of the nonlinear system. When the nonlinear behavior needs to be approximated over a range of operating conditions, we can use an array of linear models that are interconnected by suitable interpolation rules. Such a model is called an LPV model.

To generate an LPV model, the nonlinear model is trimmed and linearized over a grid of operating points. For this purpose, the operating space is parameterized by a small number of *scheduling parameters*. These parameters are often a subset of the inputs, states, and output variables of the nonlinear system. An important consideration in the creation of LPV models is the identification of a scheduling parameter set and selection of a range of parameter values at which to linearize the model.

We illustrate this approach for approximating the pitch dynamics of an airframe.

Pitch Dynamics of an Airframe

Consider a three-degree-of-freedom model of the pitch axis dynamics of an airframe. The states are the Earth coordinates (X_e, Z_e), the body coordinates (u, w), the pitch angle θ, and the pitch rate $q = \dot{\theta}$. Figure 1 summarizes the relationship between the inertial and body frames, the flight path angle γ, the incidence angle α, and the pitch angle θ.

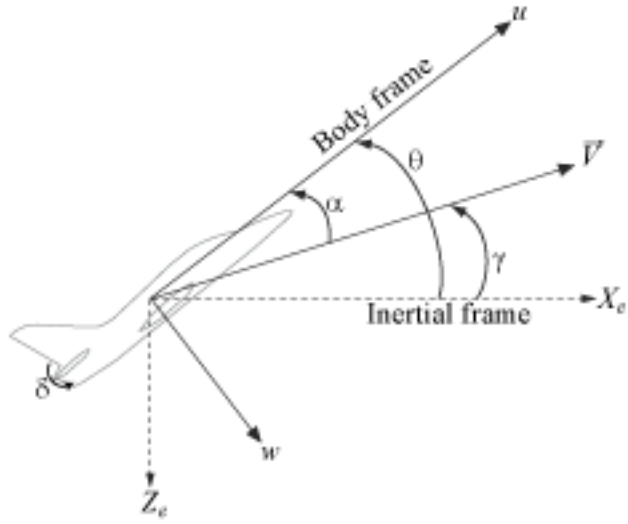
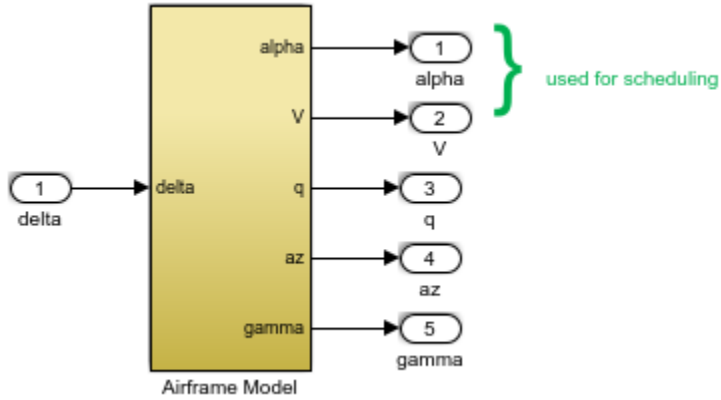


Figure 1: Airframe dynamics.

The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed V and incidence α. The model `scdairframeTRIM` describes these dynamics.

```
open_system('scdairframeTRIM')
```



Batch Linearization Across the Flight Envelope

Use the speed V and the incidence angle α as scheduling parameters; that is, trim the airframe model over a grid of α and V values. Note that these are two of the five outputs of the `scdairframeTRIM` model.

Assume that the incidence α varies between -20 and 20 degrees and that the speed V varies between 700 and 1400 m/s. Use a 15-by-12 grid of linearly spaced (α, V) pairs for scheduling:

```
nA = 15;   % number of alpha values
nV = 12;   % number of V values
alphaRange = linspace(-20,20,nA)*pi/180;
VRange = linspace(700,1400,nV);
[alpha,V] = ndgrid(alphaRange, VRange);
```

For each flight condition (α, V), linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). This requires computing the elevator deflection δ and pitch rate q that result in steady w and q.

Use `operspec` to specify the trim condition, use `findop` to compute the trim values of δ and q, and linearize the airframe dynamics for the resulting operating point. See the "Trimming and Linearizing an Airframe" example for details.

The body coordinates, (u, w), are known states for trimming. Therefore, you need to provide appropriate values for them, which you can specify explicitly. However, in this

example, let the model derive these known values based on each (α, V) pair. For each flight condition (α, V), update the values in the model and create an operating point specification. Repeat these steps for all 180 flight conditions.

```
clear op report
for ct = 1:nA*nV
   alpha_ini = alpha(ct);       % Incidence [rad]
   v_ini = V(ct);               % Speed [m/s]

   % Specify trim condition
   opspec(ct) = operspec('scdairframeTRIM');

   % Xe,Ze: known, not steady.
   opspec(ct).States(1).Known = [1;1];
   opspec(ct).States(1).SteadyState = [0;0];

   % u,w: known, w steady
   opspec(ct).States(3).Known = [1 1];
   opspec(ct).States(3).SteadyState = [0 1];

   % theta: known, not steady
   opspec(ct).States(2).Known = 1;
   opspec(ct).States(2).SteadyState = 0;

   % q: unknown, steady
   opspec(ct).States(4).Known = 0;
   opspec(ct).States(4).SteadyState = 1;

end
opspec = reshape(opspec, [nA nV]);
```

Trim the model for all of the specified ooperating point specifications.

```
Options = findopOptions('DisplayReport','off', ...
    'OptimizerType','lsqnonlin');
Options.OptimizationOptions.Algorithm = 'trust-region-reflective';
[op, report] = findop('scdairframeTRIM',opspec,Options);
```

The `op` array contains the operating points found by `findop` that will be used for linearization. The `report` array contains a record of input, output, and state values at each point.

Specify linearization inputs and outputs.

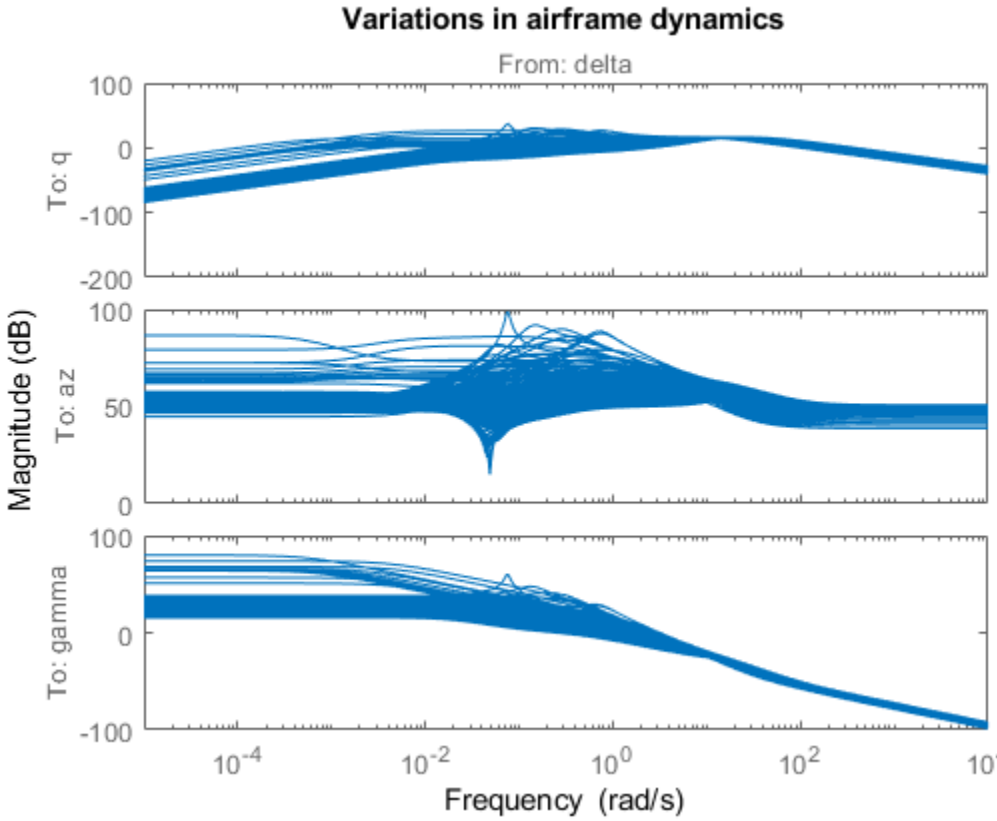
```
io = [linio('scdairframeTRIM/delta',1,'in');...         % delta
   linio('scdairframeTRIM/Airframe Model',1,'out');... % alpha
   linio('scdairframeTRIM/Airframe Model',2,'out');... % V
   linio('scdairframeTRIM/Airframe Model',3,'out');... % q
   linio('scdairframeTRIM/Airframe Model',4,'out');... % az
   linio('scdairframeTRIM/Airframe Model',5,'out')];   % gamma
```

Batch-linearize the model at the trim conditions. Store linearization offset information in the `info` structure.

```
[G,~,info] = linearize('scdairframeTRIM',op,io, ...
    linearizeOptions('StoreOffsets',true));
G = reshape(G,[nA nV]);
G.u = 'delta';
G.y = {'alpha','V','q','az','gamma'};
G.SamplingGrid = struct('alpha',alpha,'V',V);
```

`G` is a 15-by-12 array of linearized plant models at the 180 flight conditions (α, V). The plant dynamics vary substantially across the flight envelope, including scheduling locations where the local dynamics are unstable.

```
bodemag(G(3:5,:,:,:))
title('Variations in airframe dynamics')
```



The LPV System Block

The LPV System block in the Control System Toolbox™ block library facilitates simulation of linear parameter varying systems. The primary data required by the block is the state-space system array `G` that was generated by batch linearization. We augment this with information about the input/output, state, and state derivative offsets from the `info` structure.

Extract the offset information.

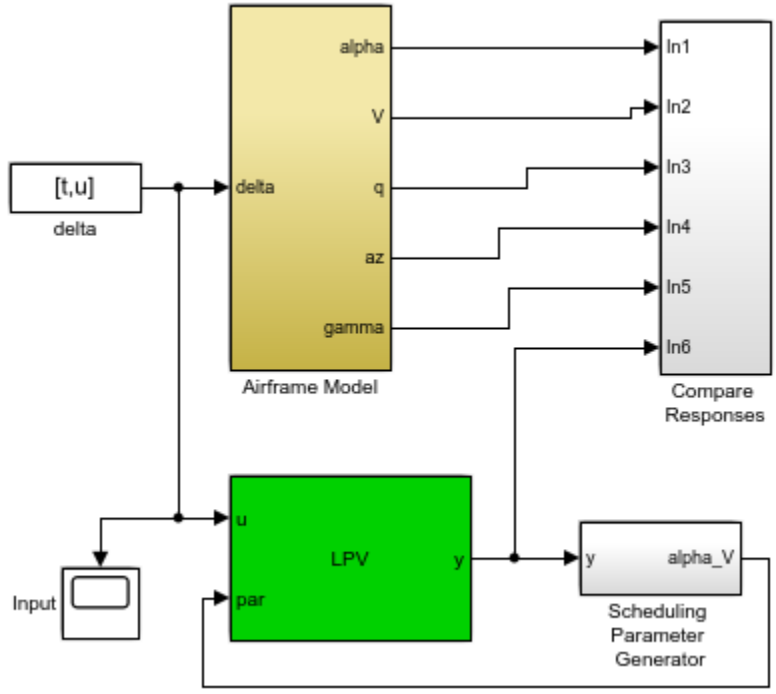
```
offsets = getOffsetsForLPV(info);
xOffset = offsets.x;
yOffset = offsets.y;
```

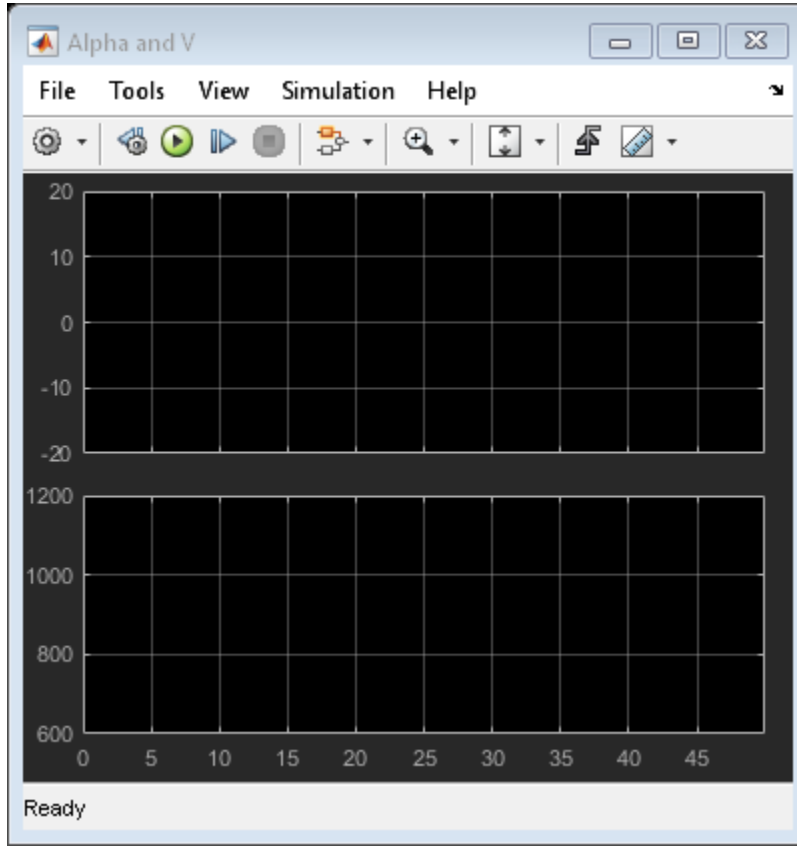
```
uOffset = offsets.u;
dxOffset = offsets.dx;
```

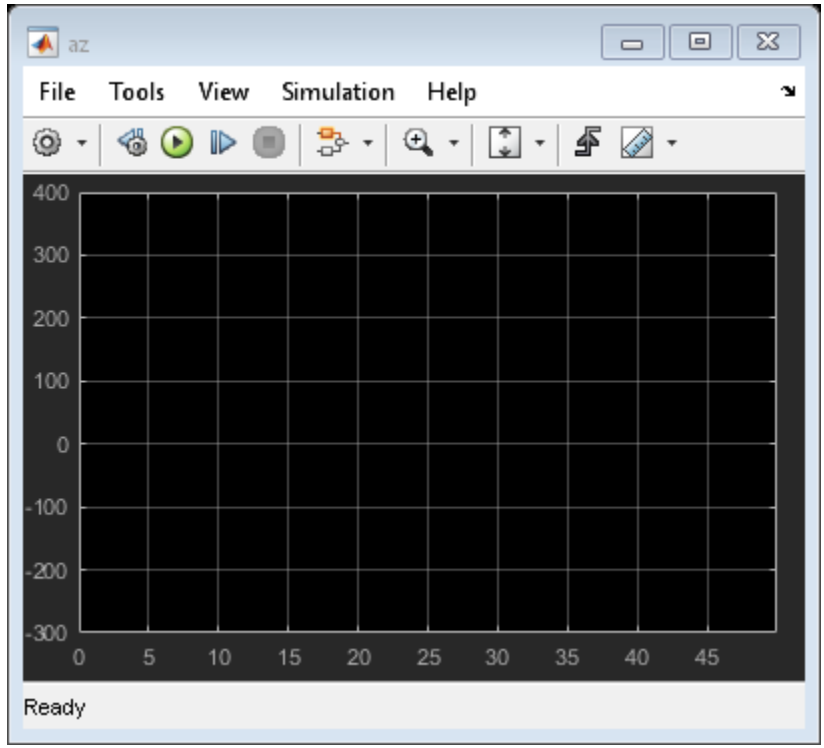
LPV Model Simulation

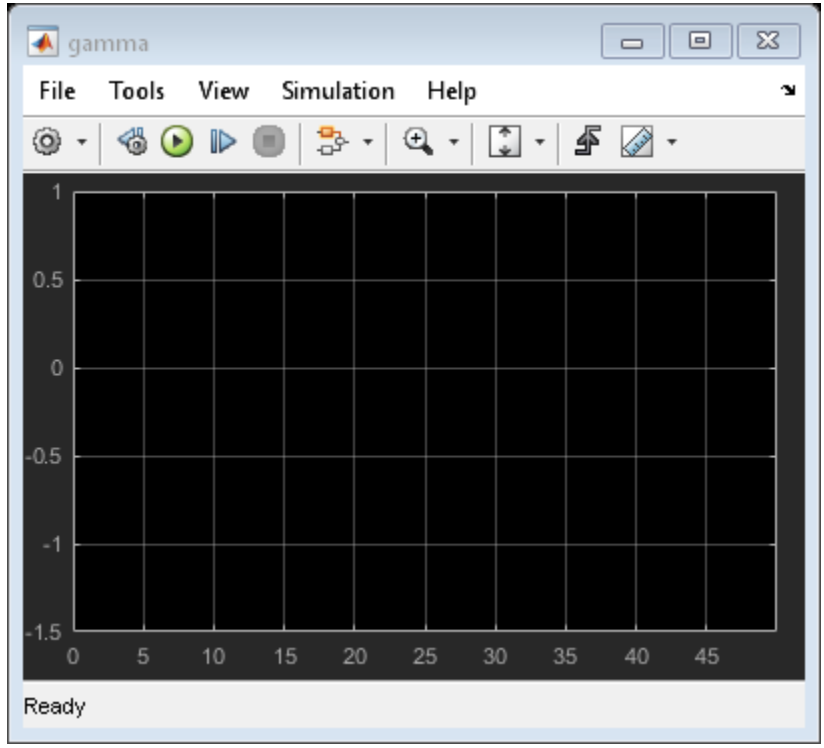
Open the system `scdairframeLPV`, which contains an LPV System block that has been configured based on linear system array `G` and the various offsets.

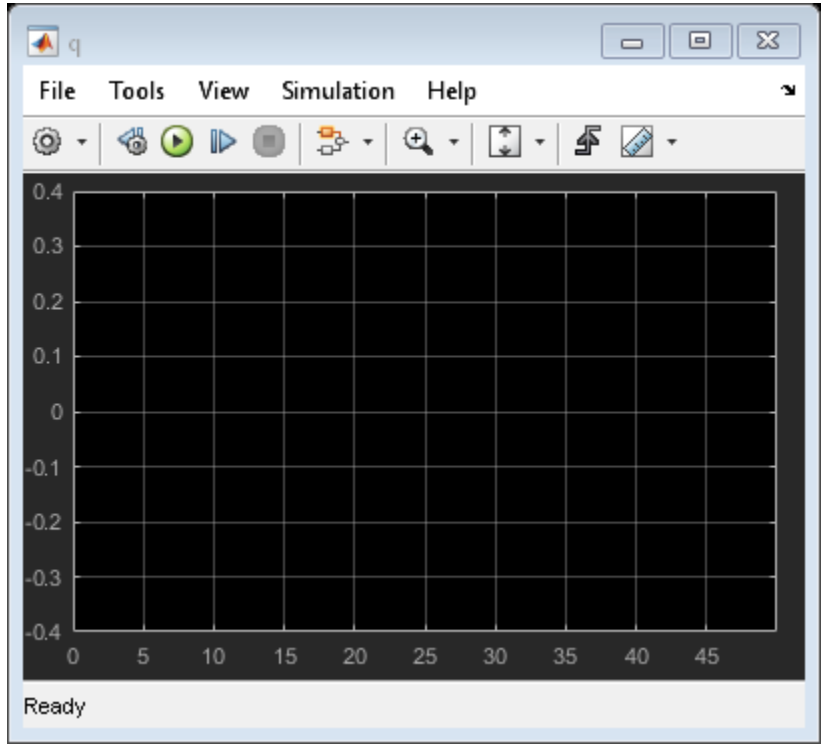
```
open_system('scdairframeLPV')
```

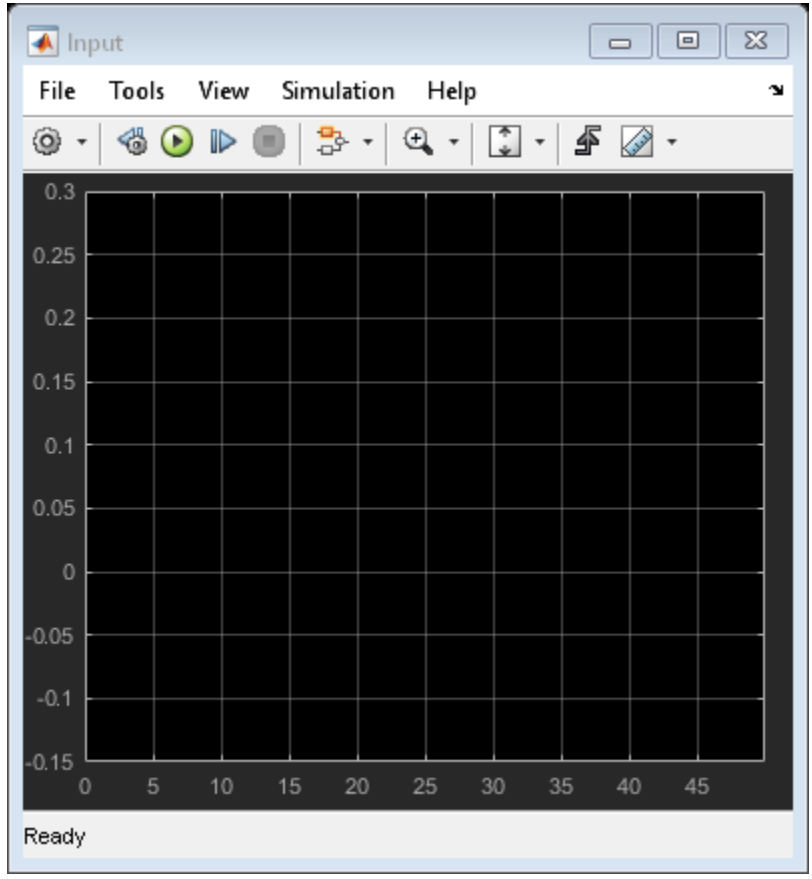



Alpha and V
File
Tools
View
Simulation
Help
20
10
0
-10
-20
1200
1000
800
600
0
5
10
15
20
25
30
35
40
45
Ready


az
File
Tools
View
Simulation
Help
400
300
200
100
0
-100
-200
-300
0
5
10
15
20
25
30
35
40
45
Ready

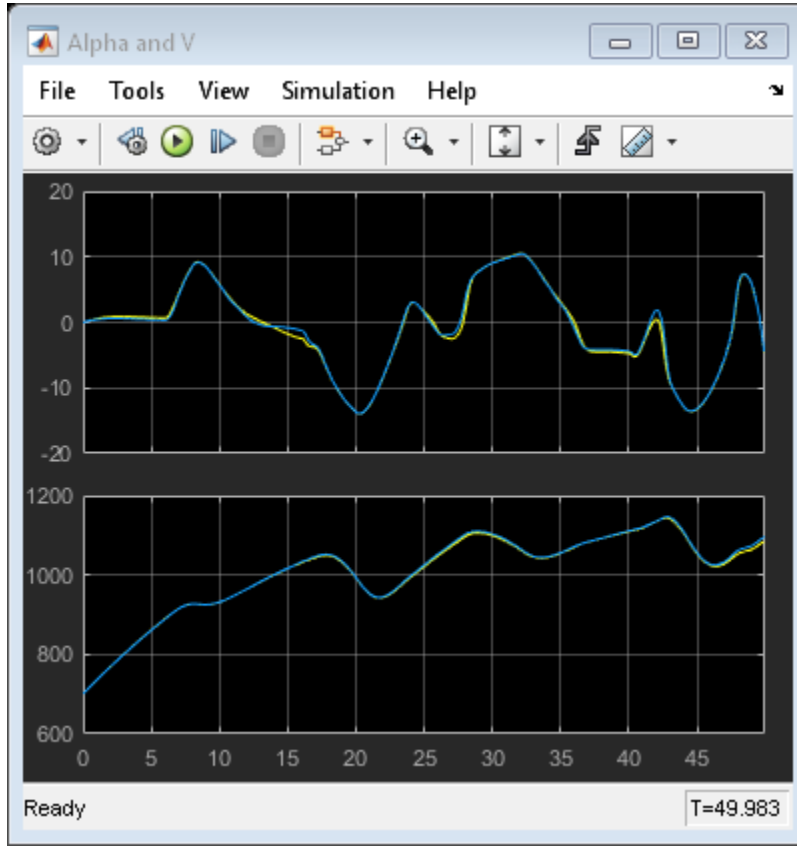

gamma
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
-1.5
0
5
10
15
20
25
30
35
40
45
Ready

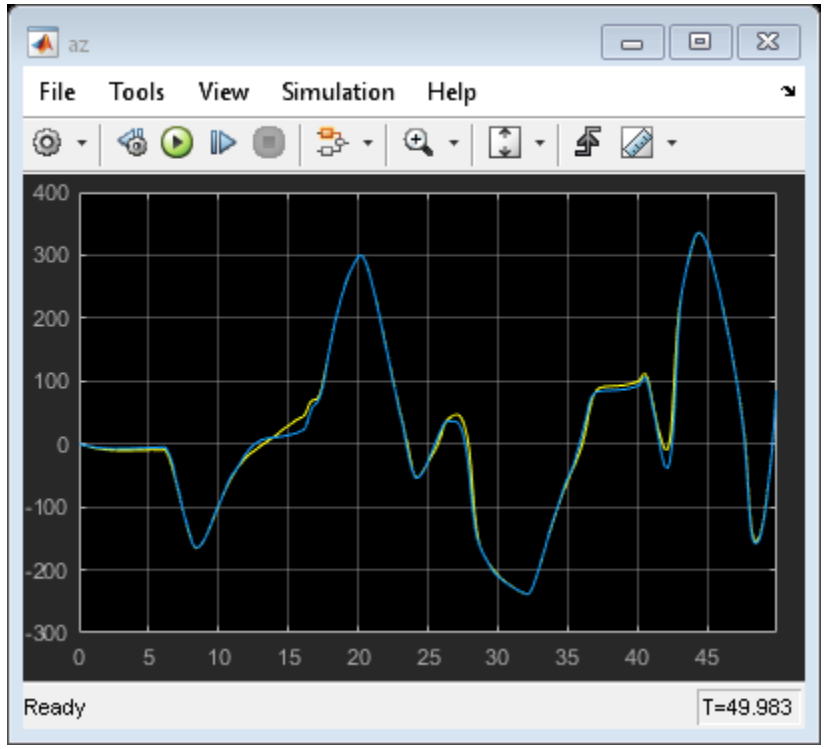

q
File
Tools
View
Simulation
Help
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
20
25
30
35
40
45
Ready

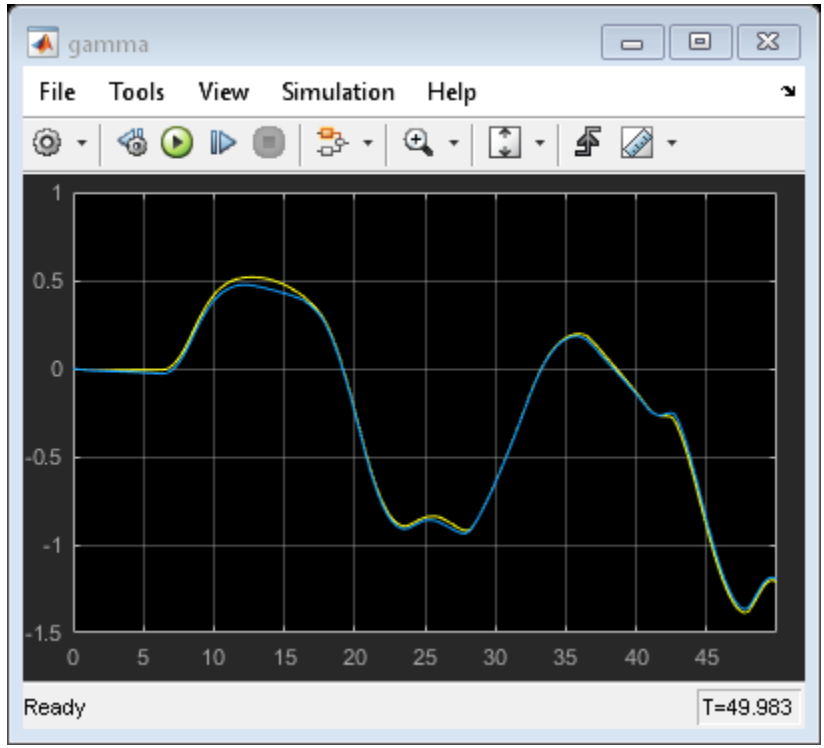


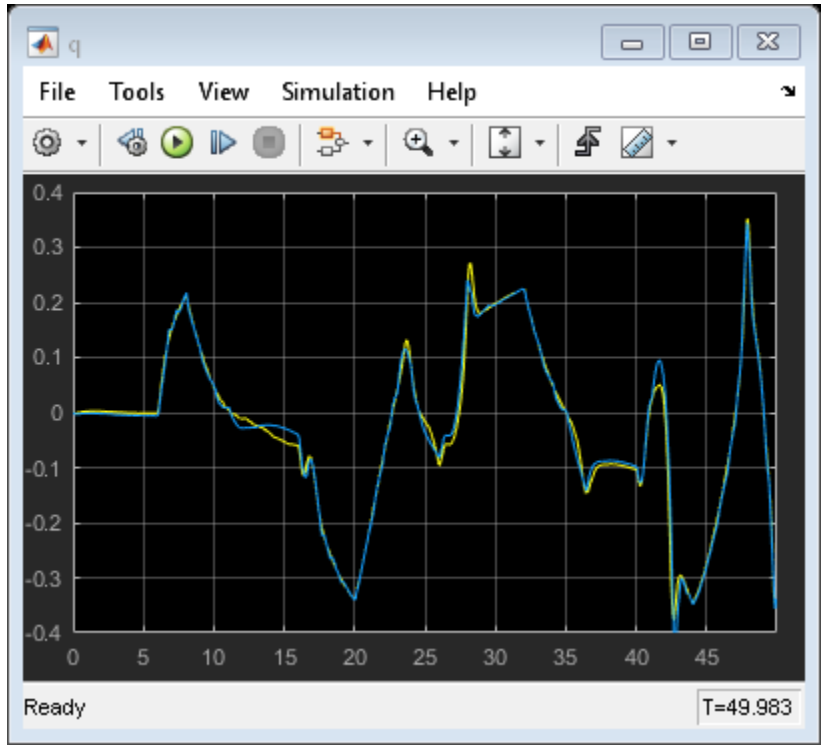
An input signal was prepared based on a desired trajectory of the airframe. This signal `u` and corresponding time vector `t` are saved in the `scdairframeLPVsimdata.mat` file. Specify the initial conditions for simulation.

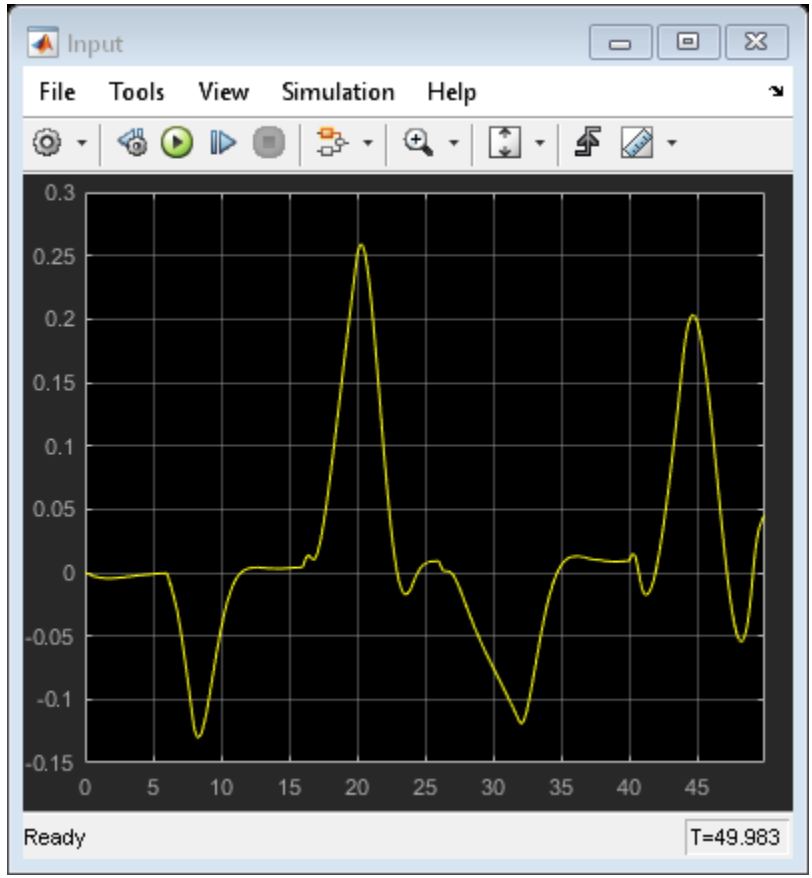
```
alpha_ini = 0;
v_ini = 700;
x0 = [0; 700; 0; 0];
sim('scdairframeLPV')
```


Alpha and V
File
Tools
View
Simulation
Help
20
10
0
-10
-20
1200
1000
800
600
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


az
File
Tools
View
Simulation
Help
400
300
200
100
0
-100
-200
-300
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


gamma
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
-1.5
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


q
File
Tools
View
Simulation
Help
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983



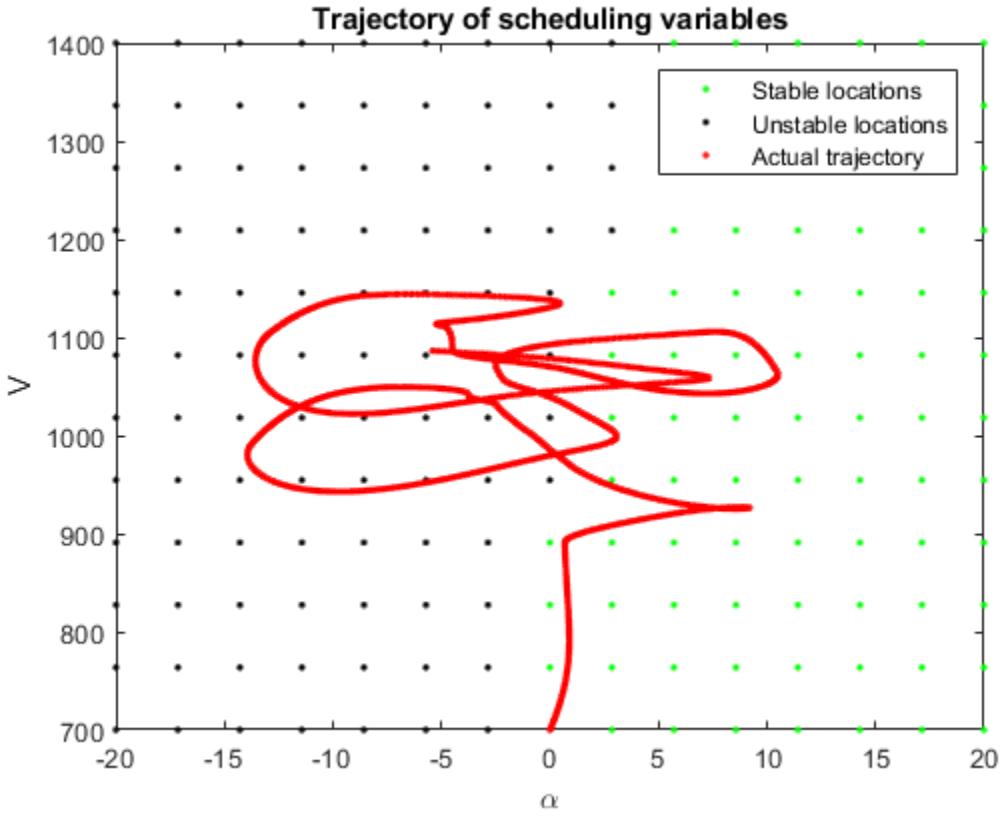
The simulation shows good emulation of the airframe response by the LPV system. We chose a very fine gridding of scheduling space leading to a large number (180) of linear models. Large array sizes can increase implementation costs. However, the advantage of LPV representations is that we can adjust the scheduling grid (and hence the number of linear systems in the array) based on:

- The scheduling subspace spanned by the anticipated trajectory
- The level of accuracy desired in an application

The former information helps reduce the range for the scheduling variables. The latter helps pick an optimal resolution (spacing) of samples in the scheduling space.

Let us plot the actual trajectory of scheduling variables in the previous simulation against the backdrop of gridded scheduling space. The (α, V) outputs were logged via their scopes (contained inside the Compare Responses block of `scdairframeLPV`).

```
Stable = false(nA,nV);
for ct = 1:nA*nV
   Stable(ct ) = isstable(G(:,:,ct));
end
alpha_trajectory = Alpha_V_Data.signals(1).values(:,1);
V_trajectory = Alpha_V_Data.signals(2).values(:,1);

plot(alpha(Stable)*180/pi,V(Stable),'g.',...
   alpha(~Stable)*180/pi,V(~Stable),'k.',...
   alpha_trajectory,V_trajectory,'r.')
title('Trajectory of scheduling variables')
xlabel('\alpha'); ylabel('V')
legend('Stable locations','Unstable locations','Actual trajectory')
```



The trajectory traced during simulation is shown in red. Note that it traverses both the stable and unstable regions of the scheduling space. Suppose you want to implement this model on a target hardware for input profiles similar to the one used for simulation above, while using the least amount of memory. The simulation suggests that the trajectory mainly stays in the 890 to 1200 m/s range of velocities and -15 to 12 degree range of incidence angle. Furthermore, you can explore increasing the spacing between the sampling points. Suppose you use only every third sample along the V dimension and every second sample along the α dimension. The reduced system array meeting these constraints can be extracted from `G` as follows:

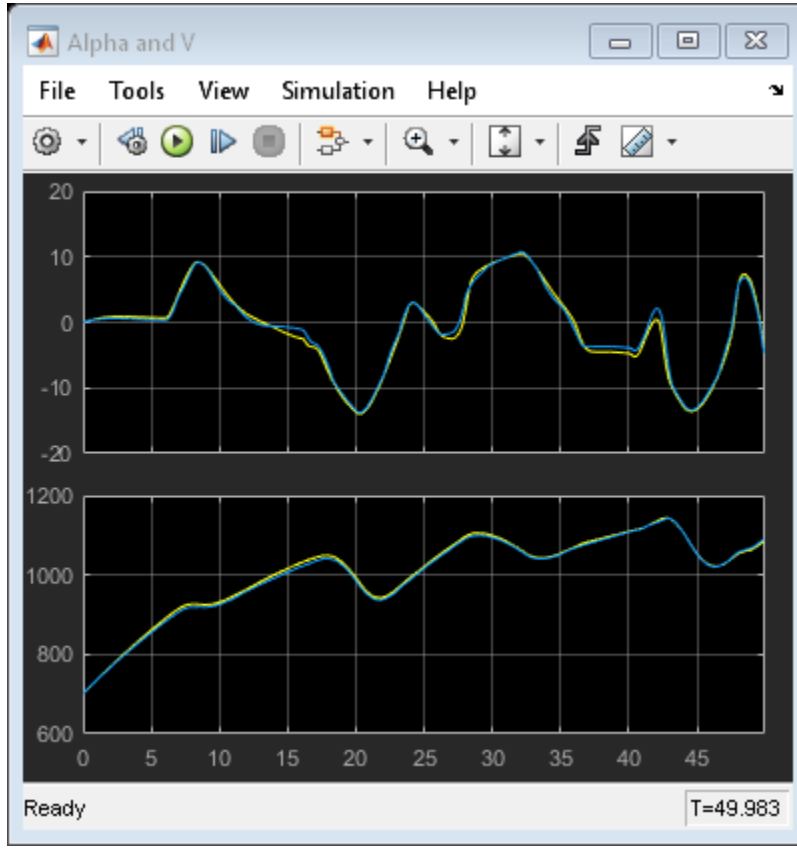
```
I1 = find(alphaRange>=-15*pi/180 & alphaRange<=12*pi/180);
I2 = find(VRange>=890 & VRange<=1200);
I1 = I1(1:2:end);
```

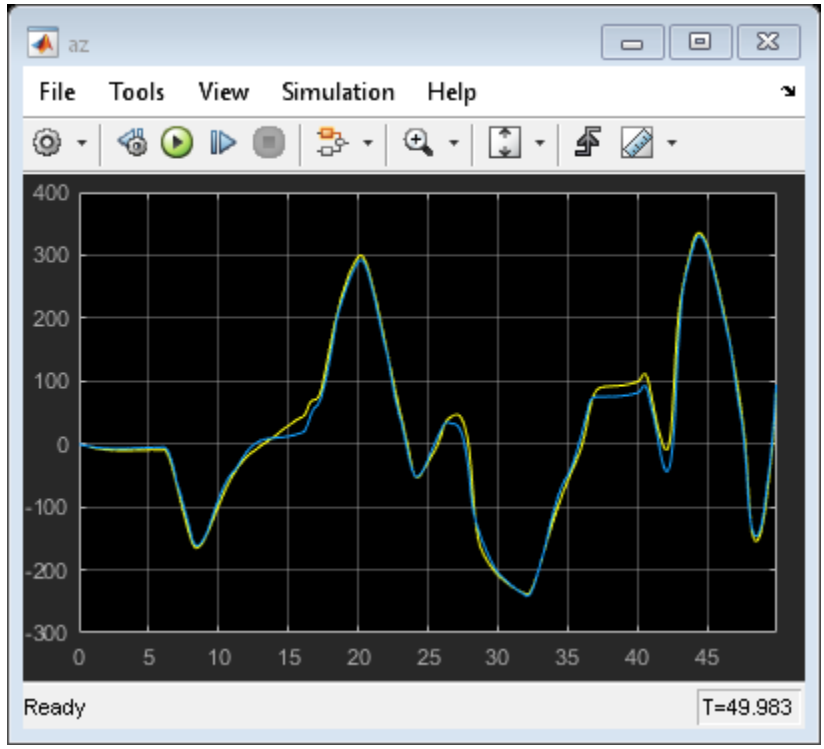
```
I2 = I2(1:3:end);

Gr = G(:,:,I1,I2);
size(Gr)

5x2 array of state-space models.
Each model has 5 outputs, 1 inputs, and 4 states.
```

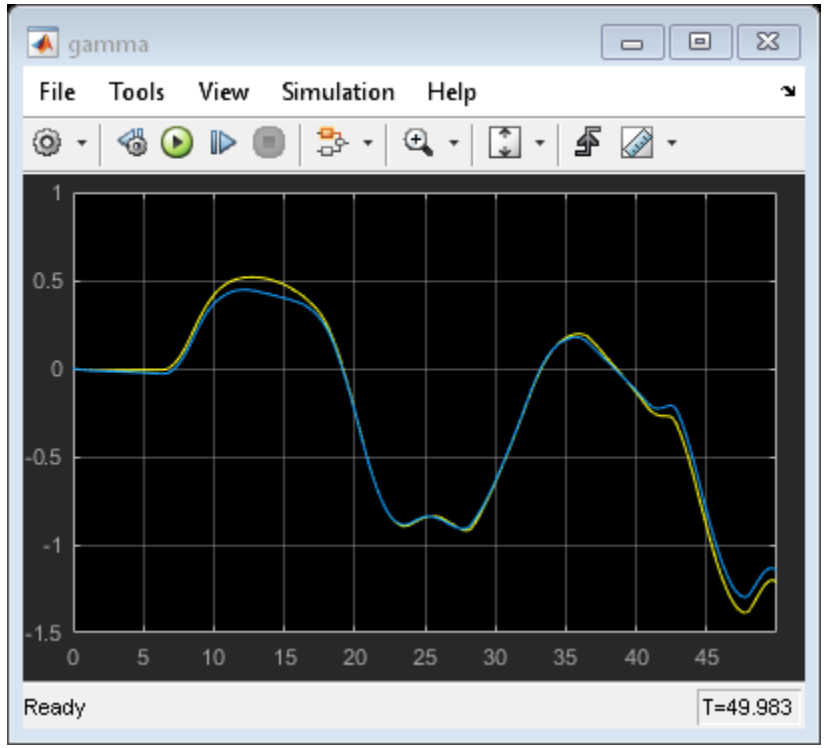
The new sampling grid, `Gr`, has a more economical size of 5-by-2. Simulate the reduced model and check its fidelity in reproducing the original behavior.

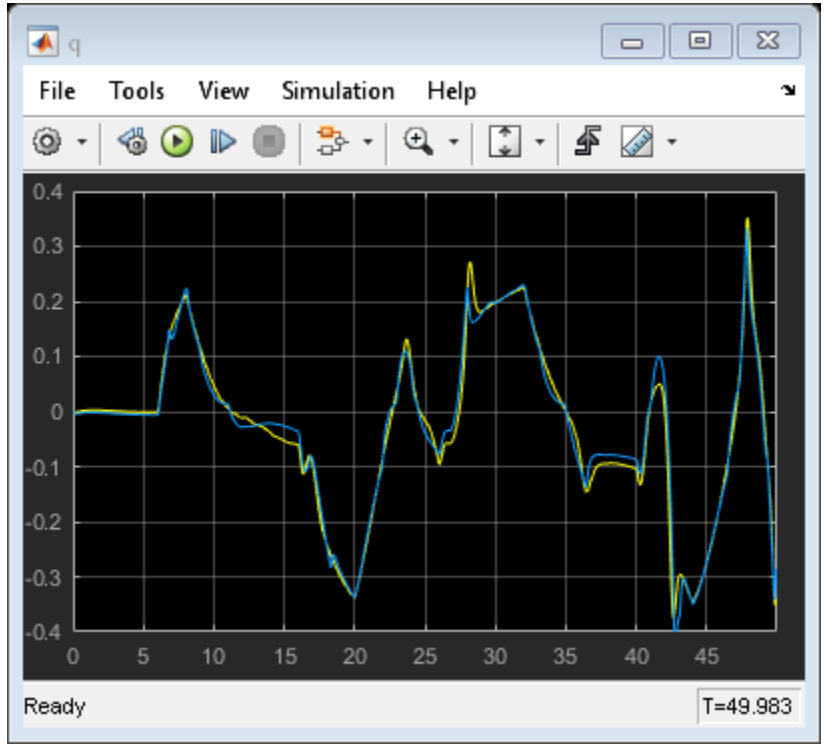
Change directory to a writable directory since model would need to be recompiled

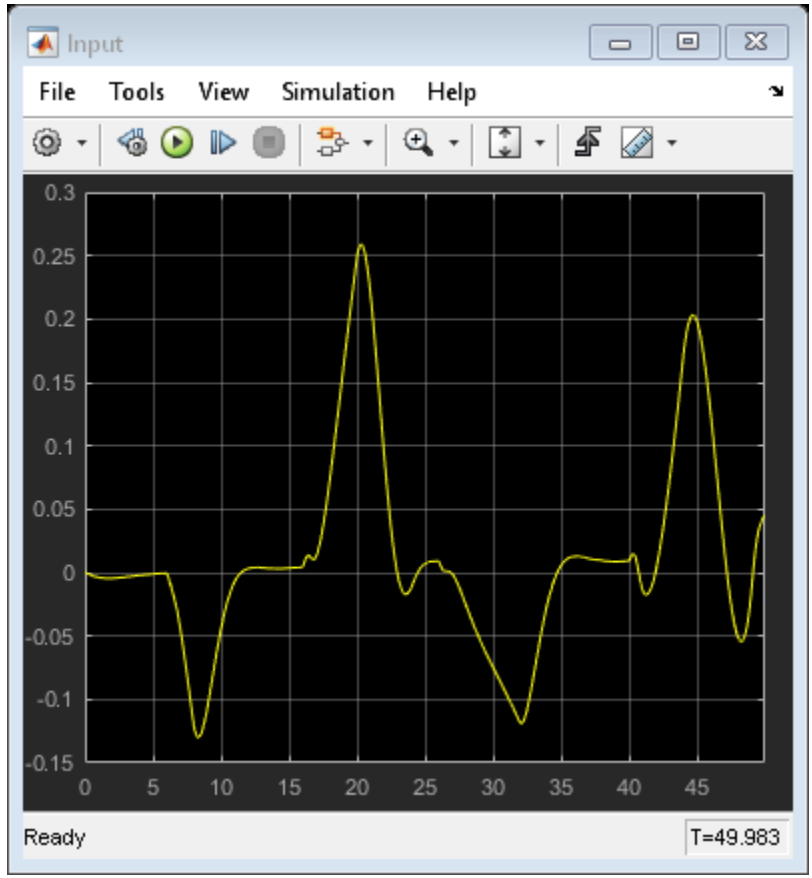
```
cwd = pwd;
cd(tempdir)
lpvblk = 'scdairframeLPV/LPV System';
set_param(lpvblk,...
   'sys','Gr',...
   'uOffset','uOffset(:,:,I1,I2)',...
   'yOffset','yOffset(:,:,I1,I2)',...
   'xOffset','xOffset(:,:,I1,I2)',...
   'dxOffset','dxOffset(:,:,I1,I2)')
sim('scdairframeLPV')
cd(cwd)
```


Alpha and V
File
Tools
View
Simulation
Help
20
10
0
-10
-20
1200
1000
800
600
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


az
File
Tools
View
Simulation
Help
400
300
200
100
0
-100
-200
-300
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


gamma
File
Tools
View
Simulation
Help
1
0.5
0
-0.5
-1
-1.5
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983


q
File
Tools
View
Simulation
Help
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
20
25
30
35
40
45
Ready
T=49.983



There is no significant reduction in overlap between the response of the original model and its LPV proxy.

The LPV model can serve as a proxy for the original system in situations where faster simulations are required. The linear systems used by the LPV model may also be obtained by system identification techniques (with additional care required to maintain state consistency across the array). The LPV model can provide a good surrogate for

initializing simulink design optimization problems and performing fast hardware-in-loop simulations.

See Also

LPV System | `linearize`

Related Examples

- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-28
- “LPV Approximation of a Boost Converter Model” on page 3-117

LPV Approximation of a Boost Converter Model

This example shows how you can obtain a Linear Parameter Varying (LPV) approximation of a Simscape Power Systems™ model of a Boost Converter. The LPV representation allows quick analysis of average behavior at various operating conditions.

Boost Converter Model

A Boost Converter circuit converts a DC voltage to another DC voltage by controlled chopping or switching of the source voltage. The request for a certain load voltage is translated into a corresponding requirement for the transistor duty cycle. The duty cycle modulation is typically several orders of magnitude slower than the switching frequency. The net effect is attainment of an average voltage with relatively small ripples. See Figure 1 for a zoomed-in view of this dynamics.

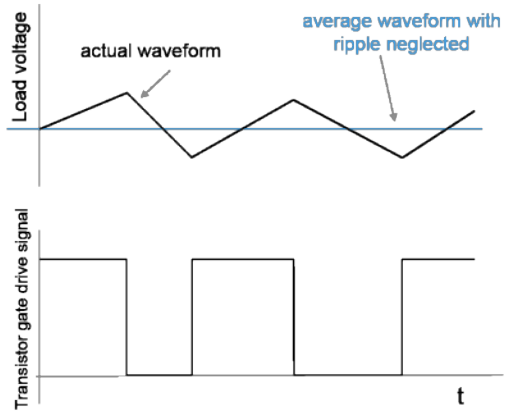


Figure 1: Converter output (load) voltage generation

In practice there are also disturbances in the source voltage V_{dc} and the resistive load R affecting the actual load voltage V_{load}.

Open the Simulink model.

```
mdl = 'BoostConverterExampleModel';
open_system(mdl);
```

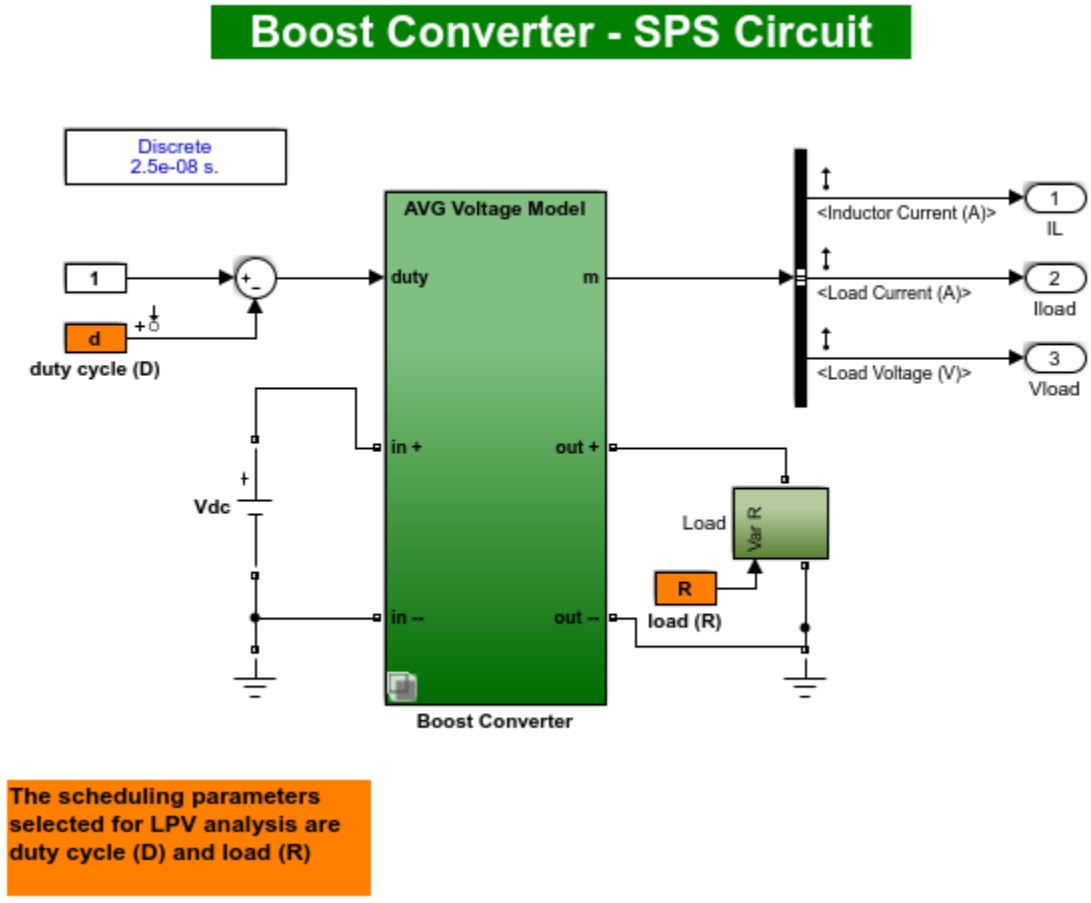


Figure 2: Simscape Power Systems based Boost Converter model

The circuit in the model is characterized by high frequency switching. The model uses a sample time of 25 ns. The "Boost Converter" block used in the model is a variant subsystem that implements 3 different versions of the converter dynamics. Double click on the block to view these variants and their implementations. The model takes the duty cycle value as its only input and produces three outputs - the inductor current, the load current and the load voltage.

The model simulates slowly (when looking for changes in say 0 - 10 ms) owing to the high frequency switching elements and small sample time.

Batch Trimming and Linearization

In many applications, the average voltage delivered in response to a certain duty cycle profile is of interest. Such behavior is studied at time scales several decades larger than the fundamental sample time of the circuit. These "average models" for the circuit are derived by analytical considerations based on averaging of power dynamics over certain time periods. The model `BoostConverterExampleModel` implements such an average model of the circuit as its first variant, called "AVG Voltage Model". This variant typically executes faster than the "Low Level Model" variant.

The average model is not a linear system. It shows nonlinear dependence on the duty cycle and the load variations. To aid faster simulation and voltage stabilizing controller design, we can linearize the model at various duty cycle and load values. The inputs and outputs of the linear system would be the same as those of the original model.

We use the snapshot time based trimming and linearization approach. The scheduling parameters are the duty cycle value (d) and the resistive load value (R). The model is trimmed at various values of the scheduling parameters resulting in a grid of linear models. For this example, we chose a span of 10%-60% for the duty cycle variation and of 4-15 Ohms for the load variation. 5 values in these ranges are picked for each scheduling variable and linearization obtained at all possible combinations of their values.

Scheduling parameters: d: duty cycle R: resistive load

```
nD = 5; nR = 5;
dspace = linspace(0.1,0.6,nD); % nD values of "d" in 10%-60% range
Rspace = linspace(4,15,nR);    % nR values of "R" in 4-15 Ohms range
[dgrid,Rgrid] = ndgrid(dspace,Rspace);  % all possible combinations of "d" and "R" valu
```

Create a parameter structure array.

```
params(1).Name = 'd';
params(1).Value = dgrid;
params(2).Name = 'R';
params(2).Value = Rgrid;
```

A simulation of the model under various conditions shows that the model's outputs settle down to their steady state values before 0.01 s. Hence we use t = 0.01s as the snapshot time.

Declare number of model inputs, outputs and states.

```
ny = 3; nu = 1; nx = 7;
ArraySize = size(dgrid);
```

Compute equilibrium operating points using `findop`. The code takes several minutes to finish.

```
op = findop(mdl, 0.01, params);
```

Get linearization input-output specified in the model.

```
io = getlinio(mdl);
```

Linearize the model at the operating point array `op` and store the offsets.

```
[linsys, ~, info] = linearize(mdl, op, io, params, ...
    linearizeOptions('StoreOffsets', true));
```

Extract offsets from the linearization results.

```
offsets = getOffsetsForLPV(info);
yoff = offsets.y;
xoff = offsets.x;
uoff = offsets.u;
```

Plot the linear system array.

```
bodemag(linsys)
grid on
```

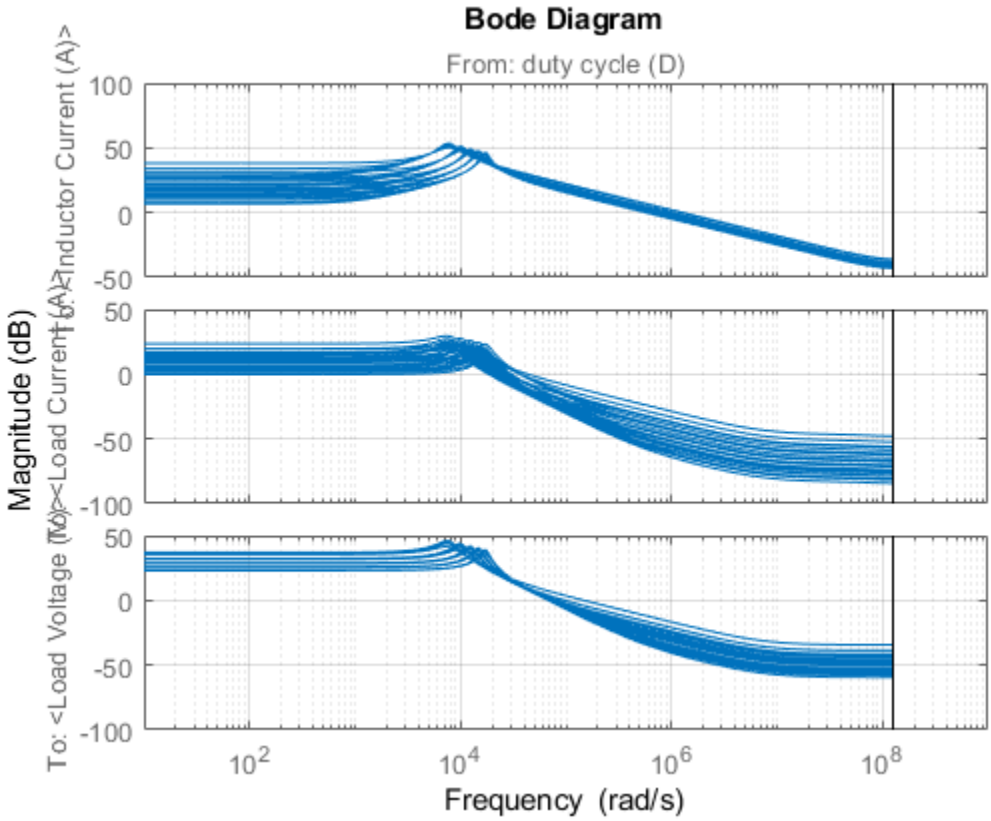


Figure 3: Bode plot of linear system array obtained over the scheduling parameter grid.

LPV Simulation: Preliminary Analysis

`linsys` is an array of 25 linear state-space models, each containing 1 input, 3 outputs and 7 states. The models are discrete-time with sample time of 25 ns. The bode plot shows significant variation in dynamics over the grid of scheduling parameters. The linear system array and the accompanying offset data (`uoff`, `yoff` and `xoff`) can be used to configure the LPV system block. The "LPV model" thus obtained serves as a linear system array approximation of the average dynamics. The LPV block configuration is available in the `BoostConverterLPVModel_Prelim` model.

```
lpvmdl = 'BoostConverterLPVModel_Prelim';
open_system(lpvmdl);
```

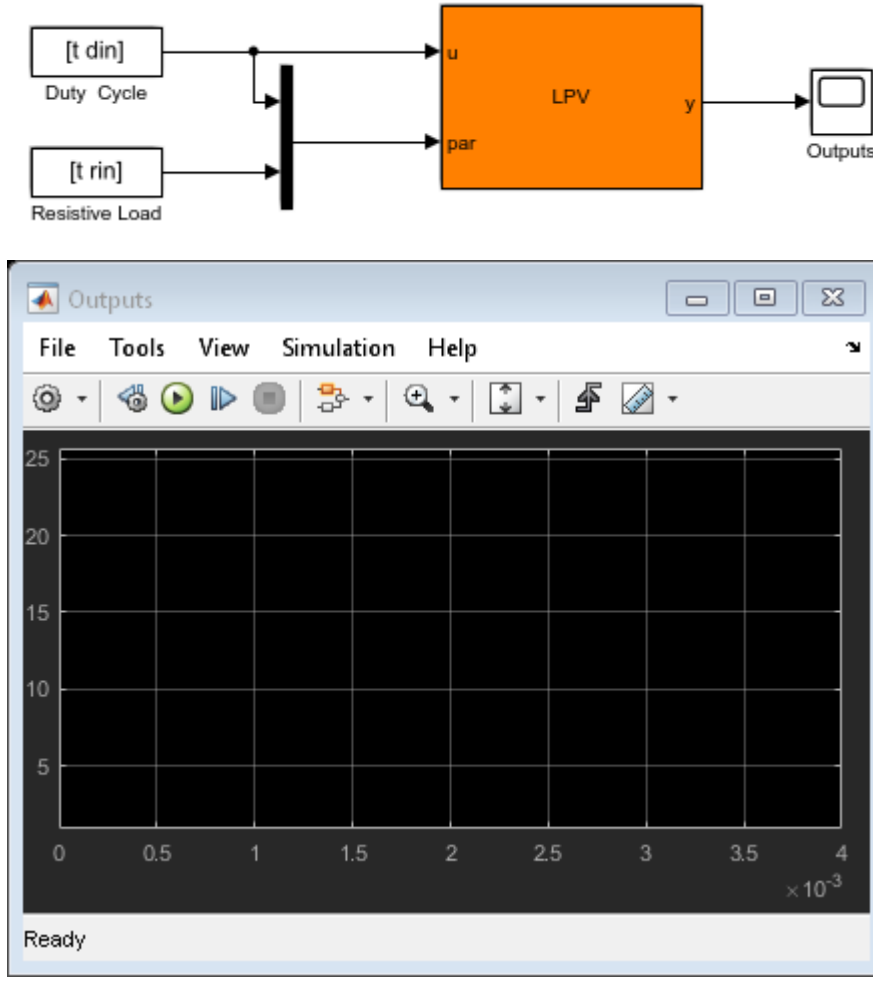


Figure 4: LPV model configured using linsys.

For simulating the model, we use an input profile for duty cycle that roughly covers its scheduling range. We also vary the resistive load to simulate the case of load disturbances.

Generate simulation data.

```
t = linspace(0,.05,1e3)';
din = 0.25*sin(2*pi*t*100)+0.25;
din(500:end) = din(500:end)+.1;  % the duty cycle profile

rin = linspace(4,12,length(t))';
rin(500:end) = rin(500:end)+3;
rin(100:200) = 6.6;              % the load profile

yyaxis left
plot(t,din)
xlabel('Time (s)')
ylabel('Duty Cycle')
yyaxis right
plot(t,rin)
ylabel('Resistive Load (Ohm)')
title('Scheduling Parameter Profiles for Simulation')
```

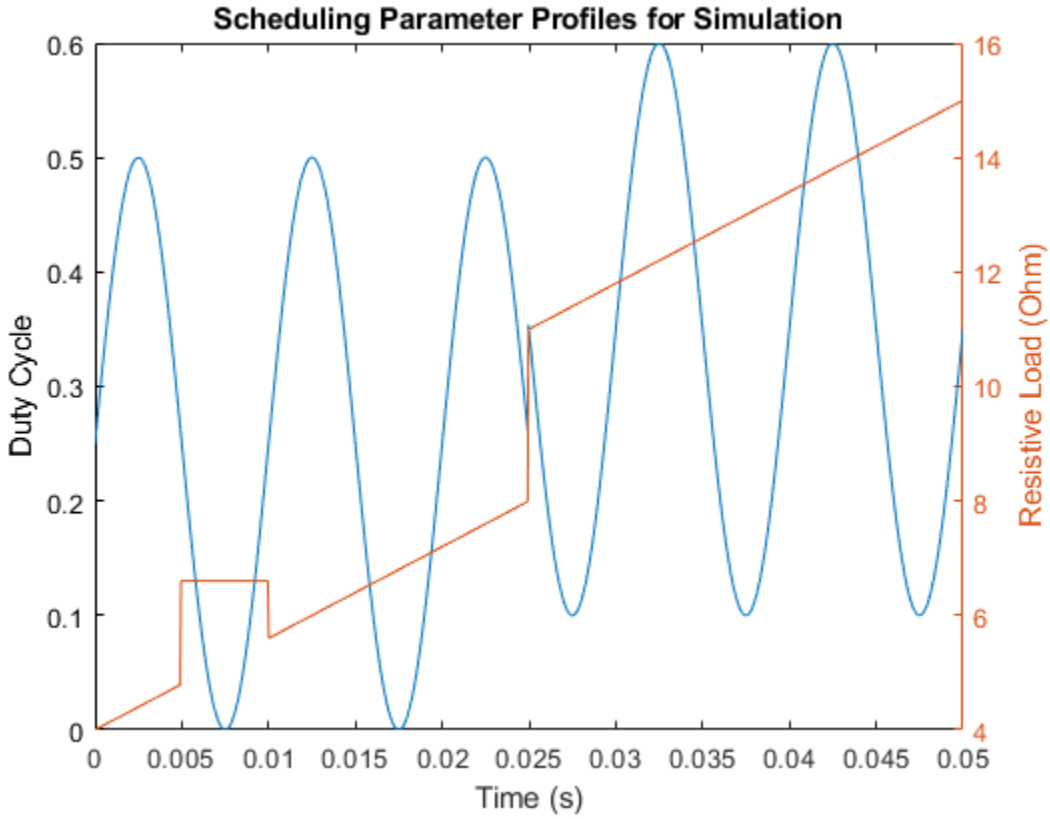


Figure 5: Scheduling parameter profiles chosen for simulation.

Note: the code for generating the above signals has been added to the model's PreLoadFcn callback for independent loading and execution. If you want to override these settings and try your own, overwrite this data in base workspace.

Simulate the LPV model.

```
sim(lpvmdl, 'StopTime', '0.004');
```

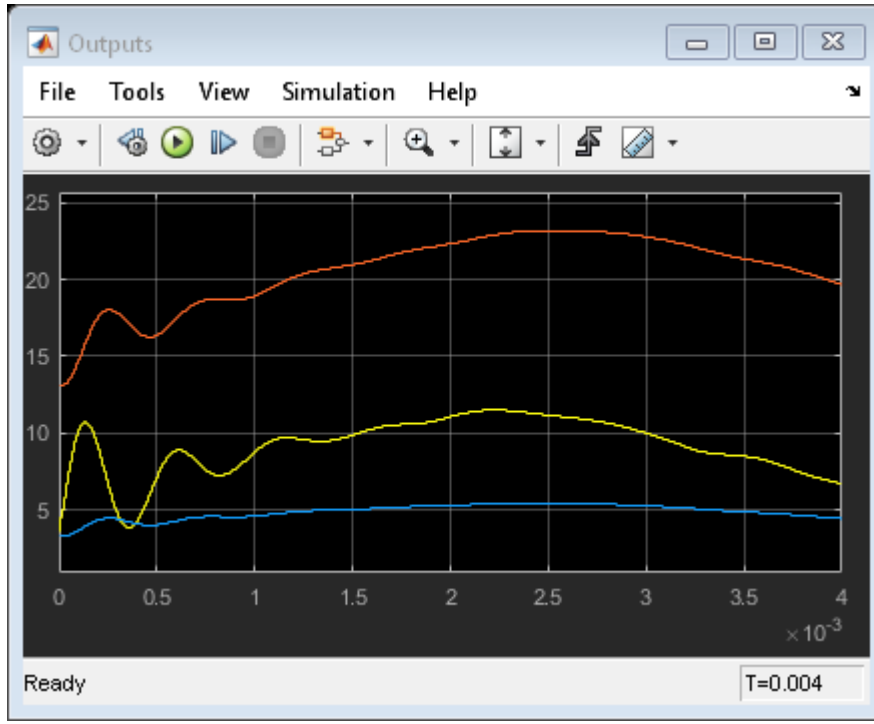


Figure 6: LPV simulation results.

Simulation shows that the LPV model is slow to simulate. So next we consider some simplifications of the LPV model. The simplifications commonly applied to linear systems involve reducing the number of states (see `balred`), modifying the sample time (see `d2d`) and removing unwanted input or output channels. In order to extend this type of analysis to LPV systems, we make the following approximation: if the scheduling parameters are assumed to be changing slowly and the system always stays close to equilibrium conditions, we can replace the model's state variables with "deviation states" $\delta x(t) = x(t) - x_{off}(t)$. With this approximation, the state offset data and initial state value can be replaced by zero values. The resulting system of equations are linear in deviation states $\delta x(t)$.

Model Order Reduction

Let us evaluate the contribution of the linear system states to the system energy across the array.

```
HSV = zeros(nx,nD*nR);
for ct = 1:nD*nR
   HSV(:,ct) = hsvd(linsys(:,:,ct));
end
ax = gca;
cla(ax,'reset');
bar3(ax, HSV)
view(ax,[-69.5 16]);
xlabel(ax, 'System Number')
ylabel(ax, 'State Number')
zlabel(ax, 'State Energy')
```

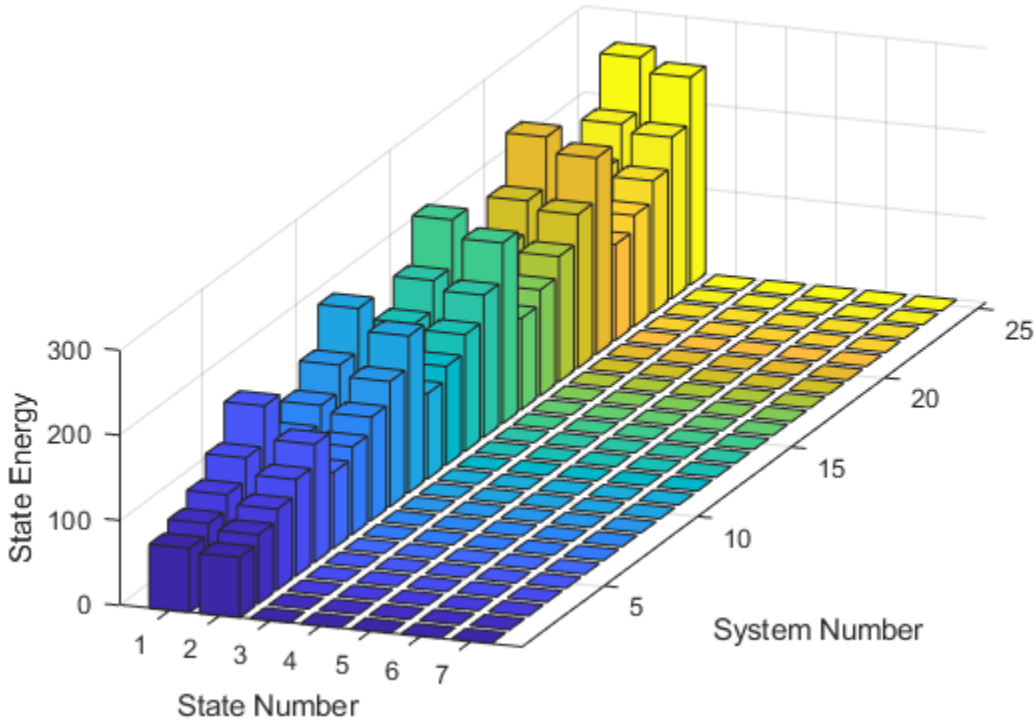


Figure 7: Bar chart of Hankel Singular Values of the linear system array linsys. The 5-by-5 array has been flattened into a 25 element system vector for plotting.

The plot shows that only 2 states are required to capture the most significant dynamics. We use this information to reduce the 7-state system array `linsys` to a 2-state array using `balred`. Here we assume that the same two transformed states contribute across the whole operating grid. Note that the LPV representation requires state-consistency across the linear system array.

```
opt = balredOptions('StateElimMethod','Truncate');
linsys2 = linsys;
for ct = 1:nD*nR
   linsys2(:,:,ct) = balred(linsys(:,:,ct),2,opt);
end
```

Upsampling the Model Array to Desired Time Scale

Next we note that the model array `linsys` (or `linsys2`) has a sample time of 25ns. We need the model to study the changes in outputs in response to duty cycle and load variations. These variations are much slower than the system's fundamental sample time and occur in the microsecond scale (0 - 50 ms). Hence we increase the model sample time by a factor of 1e4.

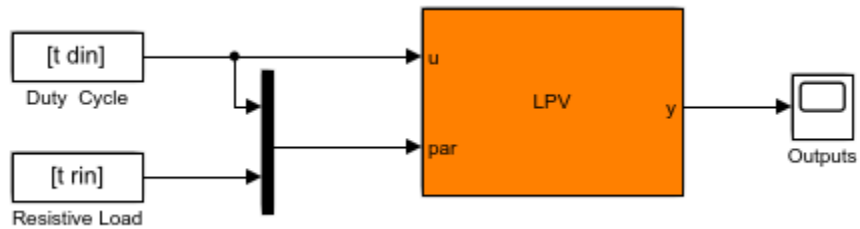
```
linsys3 = d2d(linsys2, linsys2.Ts*1e4);
```

We are now ready to make another attempt at LPV model assembly using the linear system array `linsys3` and offsets `yoff`, and `uoff`.

LPV Simulation: Final

The preconfigured model `BoostConverterLPVModel_Final` uses `linsys3` and the accompanying offset data to simulate the LPV model. It uses zero values for the state offset.

```
lpvmdl = 'BoostConverterLPVModel_Final';
open_system(lpvmdl);
sim(lpvmdl);
```



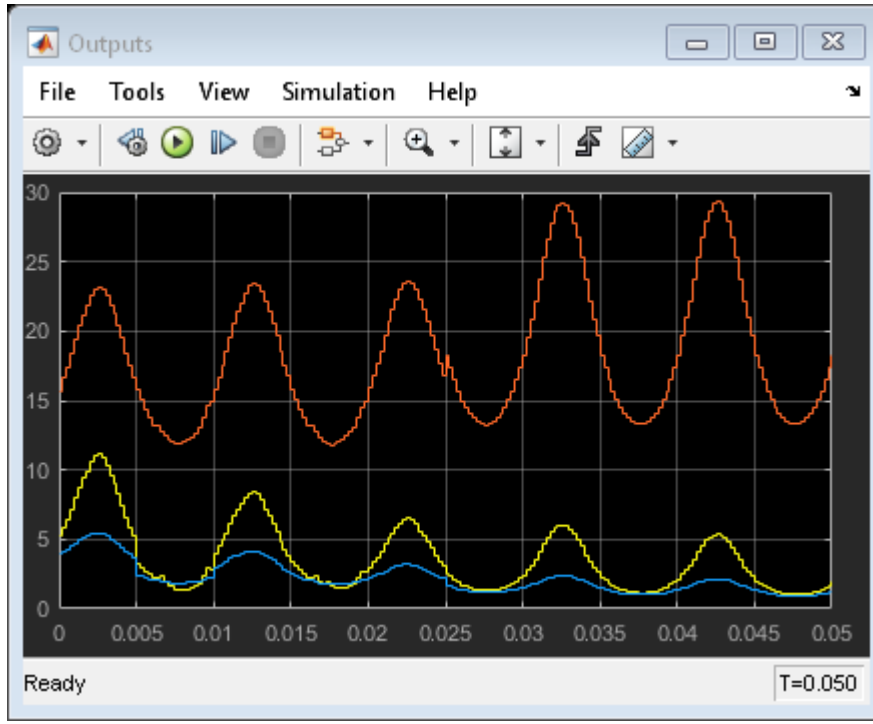


Figure 8: LPV model simulation using the reduced/scaled linear system array linsys3.

The LPV model simulates significantly faster than the original model `BoostConverterExampleModel`. But how do the results compare against those obtained from the original boost converter model? To check this, open model `BoostConverterResponseComparison`. This model has Boost Converter block configured to use the high-fidelity "Low Level Model" variant. It also contains the LPV block whose outputs are superimposed over the outputs of the boost converter in the three scopes.

```
mdl = 'BoostConverterResponseComparison';
open_system(mdl);
% sim(mdl); % uncomment to run
```

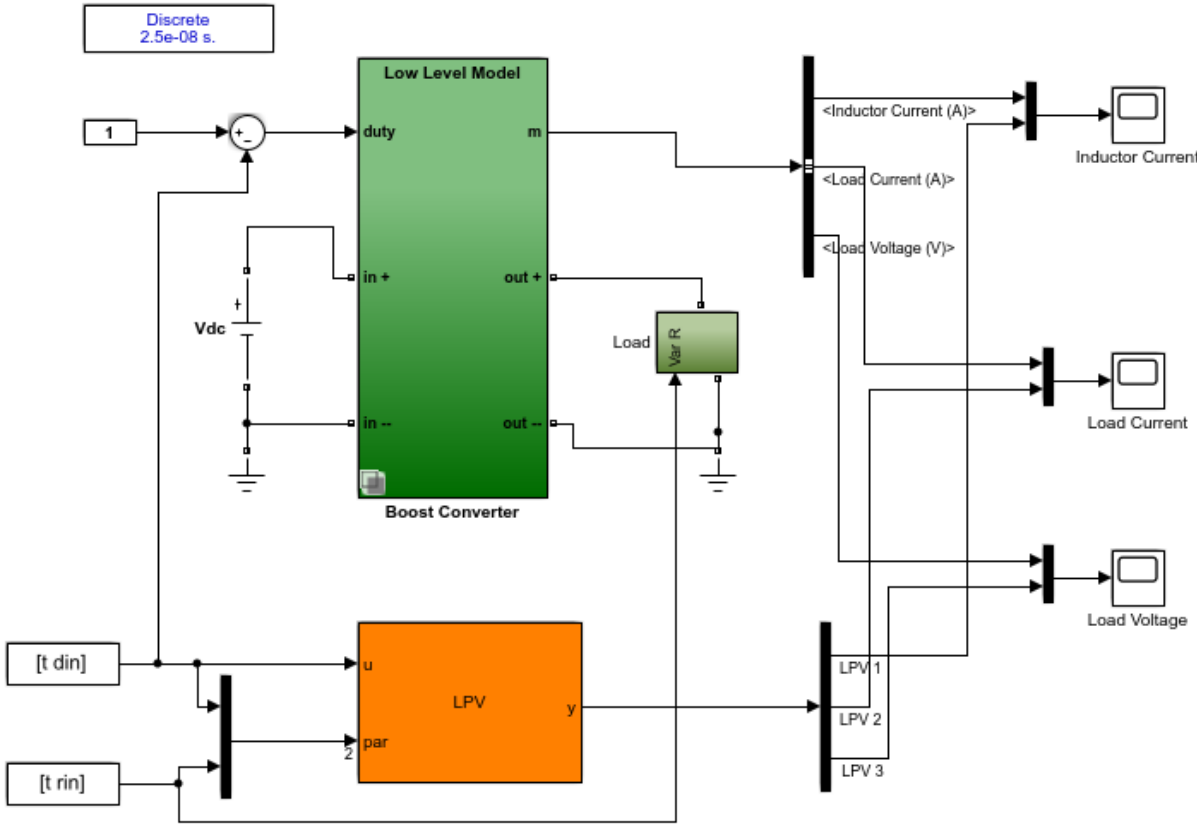


Figure 9: Model used for comparing the response of high fidelity model with the LPV approximation of its average behavior.

The simulation command has been commented out; uncomment it to run. The results are shown in the scope snapshots inserted below.

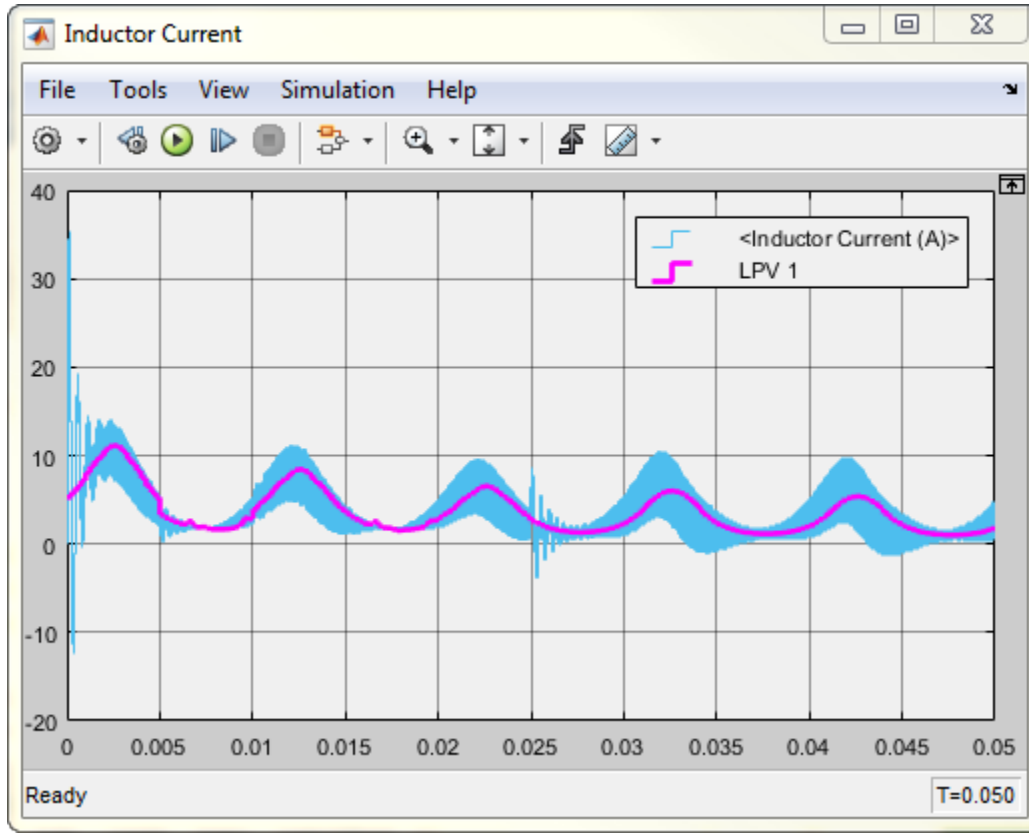


Figure 10: Inductor current signals. Blue: original, Magenta: LPV system response

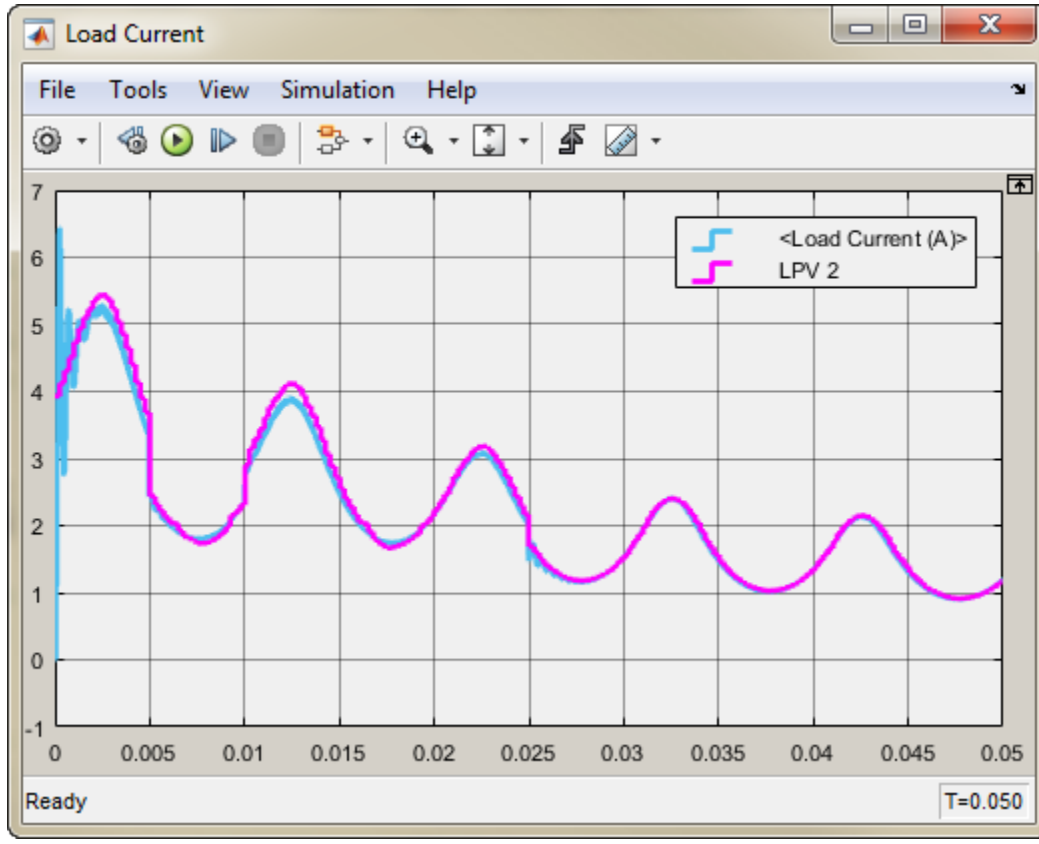


Figure 11: Load current signals. Blue: original, Magenta: LPV system response

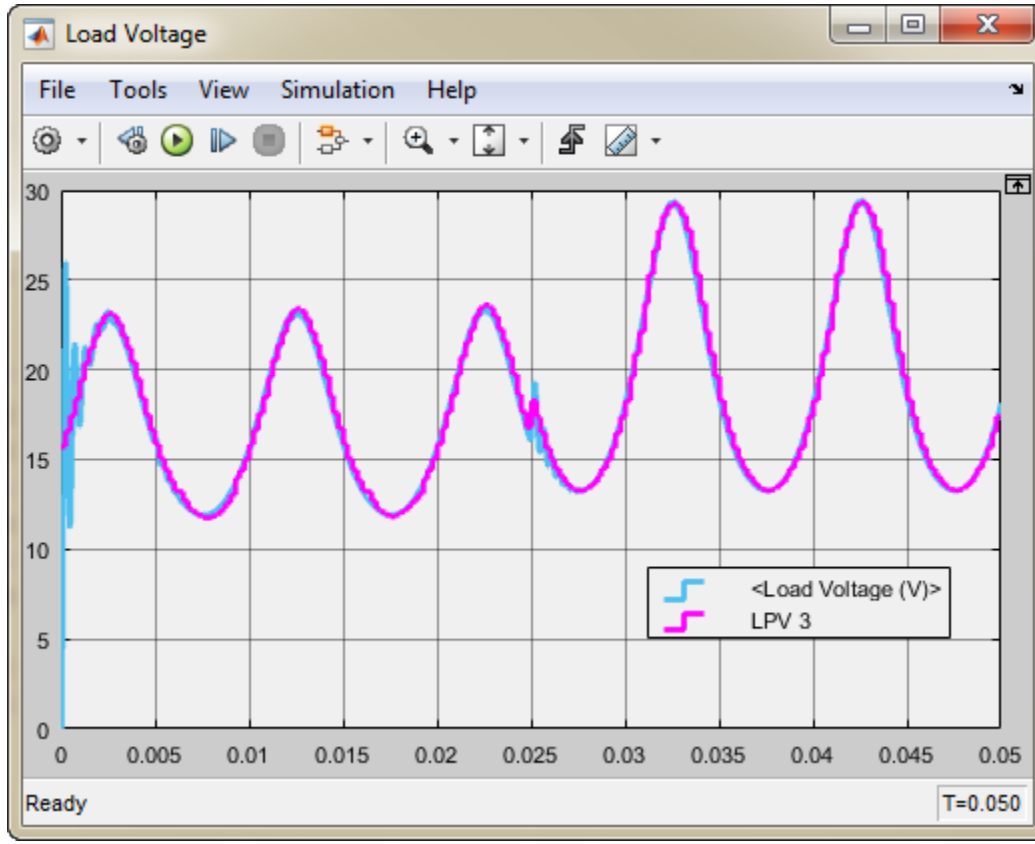


Figure 12: Load voltage signal. Blue: original, Magenta: LPV system response

The simulation runs quite slowly due to the fast switching dynamics in the original boost converter circuit. The results show that the LPV model is able to capture the average behavior nicely.

Conclusions

By using the duty cycle input and the resistive load as scheduling parameters, we were able to obtain linear approximations of average model behavior in the form of a state-space model array. This model array was further simplified by model reduction and sample rate conversion operations.

The resulting model array together with operating point related offset data was used to create an LPV approximation of the nonlinear average behavior. Simulation studies show that the LPV model is able to emulate the average behavior of a high-fidelity Simscape Power Systems model with good accuracy. The LPV model also consumes less memory and simulates significantly faster than the original system.

See Also

LPV System | `linearize`

Related Examples

- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-28
- “Approximating Nonlinear Behavior Using an Array of LTI Systems” on page 3-91

Troubleshooting Linearization Results

Linearization Troubleshooting Overview

If you do not get expected results when you linearize your Simulink model, you can diagnose and fix potential linearization issues using Simulink Control Design troubleshooting tools. The definition of an *expected* linearization result depends on your specific application.

Troubleshooting Workflow

To determine whether a linearization is successful and find potential linearization issues, first check the equations and response plots of the linearized model.

Result to Check	Signs of Successful Linearization	Signs of Unsuccessful Linearization	More Information
Linear analysis plots	Time-domain and frequency-domain response plot characteristics, such as rise time and bandwidth respectively, capture the expected dynamics of your system.	Response plot characteristics do not capture the dynamics of your system. For example: • Bode plot gain is too large or too small. • Pole-zero plot contains unexpected poles or zeros.	"Analyze Results Using Linear Analysis Tool Response Plots" on page 2-146.
Linear model equations	• State-space matrices have expected number of states, inputs, and outputs. The linearized model can have fewer states than your Simulink model because, often, the path between linearization input and output points does not reach all the model states. • Poles and zeros are in correct locations.	• Zero linearization (`D = 0`) • Infinite linearization (`D = Inf`)	"View Linearized Model Equations Using Linear Analysis Tool" on page 2-144

If the response plots or model equations of the linearized system do not capture the expected dynamics of your system, check the:

- Operating point at which you linearized the model. For more information, see “Check Operating Point” on page 4-5.
- Analysis point placement in your model. For more information, see “Check Analysis Point Placement” on page 4-6.

Once you verify that the model operating point and analysis points are correct, if your model still does not linearize as expected, you can troubleshoot the linearization results using the Linearization Advisor. The Linearization Advisor is a troubleshooting tool that allows you to identify blocks in your model that are potentially problematic for linearization. For more information, see “Identify and Fix Common Linearization Issues” on page 4-8.

Once you have identified potentially problematic blocks, you can then troubleshoot the linearizations of the individual blocks using the Linearization Advisor. For more information, see “Block Linearization Troubleshooting” on page 4-58.

Troubleshoot Linearizations of Models with Special Characteristics

Some Simulink models and blocks do not linearize well or require special considerations during linearization.

Model Characteristic	Linearization Considerations	More Information
Large models	For some large complex models, you can systematically linearize specific model components. You can then check if these components linearize as expected.	“Specify Portion of Model to Linearize” on page 2-13
Models with delays	The method with which you represent time delays in your model can affect linearization results. For example, if a Bode plot shows insufficient lag in phase, the cause can be the Padé approximation of the model time delays.	• “Models with Time Delays” on page 2-168 • “Linearization of Models with Delays” on page 2-99
Multirate models	Incorrect sample time and rate conversion methods can cause poor linearization results in multirate models.	“Linearize Multirate Models” on page 2-170

Model Characteristic	Linearization Considerations	More Information
Models with PWM signals	Models with pulse width modulation signals do not linearize well due to their discontinuities and high-frequency switching components. Consider specifying a custom linearization for such blocks.	"Configure Models with Pulse Width Modulation (PWM) Signals" on page 2-192
Models with Model Reference blocks	Linearization is not fully compatible with model reference blocks running in accelerator simulation mode. Configure these subsystems to run in normal mode during linearization.	"Linearization of Models with Model References" on page 2-106
Simscape networks	Simscape networks commonly linearize to zero when a set of the system equation Jacobians are zero at a given operating condition.	"Linearize Simscape Networks" on page 2-194

See Also

Apps

Linear Analysis Tool

More About

- "Identify and Fix Common Linearization Issues" on page 4-8
- "Block Linearization Troubleshooting" on page 4-58

Check Operating Point

To diagnose whether you used the correct operating point for linearization, simulate the model at the operating point you used for linearization.

The linearization operating point is incorrect when the critical signals in the model:

- Have unexpected values.
- Are not at steady state.

To fix the problem, compute a steady-state operating point, and repeat the linearization at this operating point. For more information, see “Compute Steady-State Operating Points” on page 1-6 and “Simulate Simulink Model at Specific Operating Point” on page 1-83.

See Also

More About

- “About Operating Points” on page 1-2
- “View and Modify Operating Points” on page 1-10

Check Analysis Point Placement

Incorrect placement of analysis points, including linearization I/Os and loop openings, can result in blocks being inappropriately included in or excluded from the linearization result linearization.

Check Linearization I/O Points Placement

After linearizing the model, check the block linearization values to determine which blocks are included in the linearization.

Blocks can be missing from the linearization path for different reasons.

Incorrect placement linearization I/O points can result in inappropriately excluded blocks from linearization. To fix the problem, specify correct linearization I/O points and repeat the linearization. For more information, see “Specify Portion of Model to Linearize” on page 2-13.

Blocks that linearize to zero (and other blocks on the same path) are excluded from linearization. To fix this problem, troubleshoot linearization of individual blocks, as described in “Block Linearization Troubleshooting” on page 4-58.

Check Loop Opening Placement

Incorrect loop opening placement causes unwanted feedback signals in the linearized model.

To fix the problem, check the individual block linearization values to identify which blocks are included in the linearization. If undesired blocks are included, place the loop opening on a different signal and repeat the linearization.

See Also

More About

- “Block Linearization Troubleshooting” on page 4-58
- “Opening Feedback Loops” on page 2-17

- “How the Software Treats Loop Openings” on page 2-39

Identify and Fix Common Linearization Issues

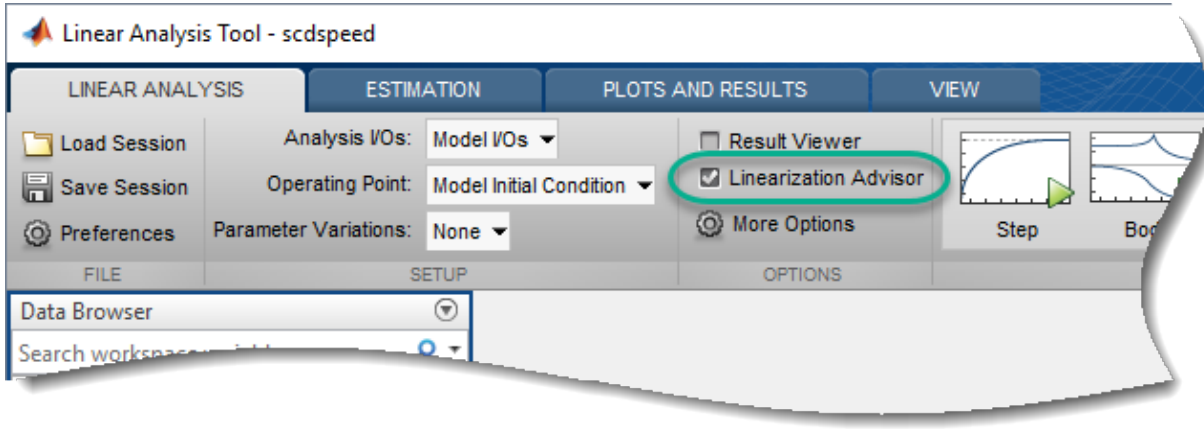
If your linearization results are not as expected, you can identify common linearization issues using the Linearization Advisor. The Linearization Advisor collects diagnostic information regarding individual block linearizations. Using this information, you can:

- View linearization details and operating points for each linearized block in your model.
- Identify potentially problematic blocks that cause common linearization issues.
- Determine which blocks are on and off the linearization path and which blocks contribute to the model linearization result.
- Search linearization results for blocks that meet specified criteria.

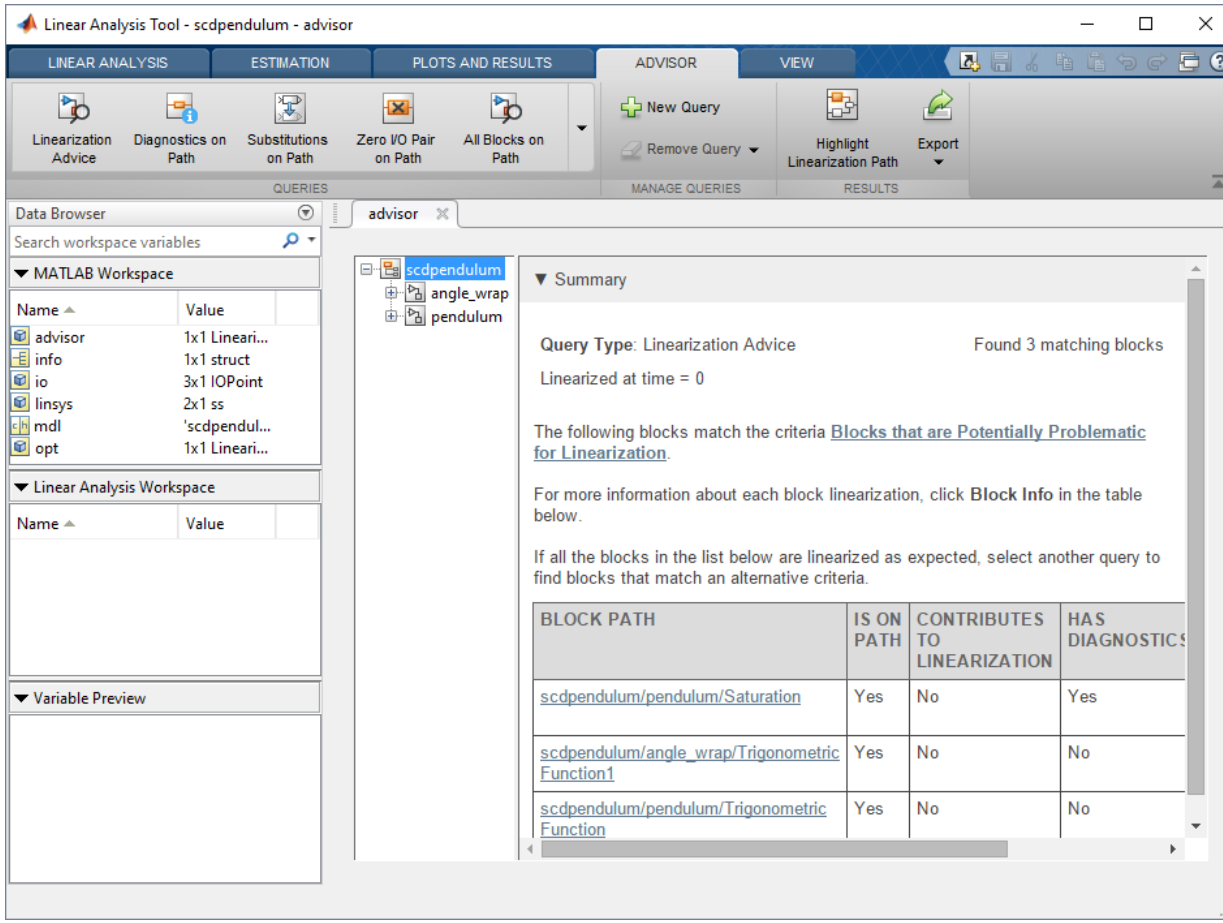
Enable Linearization Advisor

Since collecting diagnostic information adds linearization overhead, the Linearization Advisor is disabled by default. To collect diagnostic information, you must enable the Linearization Advisor before you linearize your model.

To enable the Linearization Advisor, in the Linear Analysis Tool, on the **Linear Analysis** tab, select **Linearization Advisor**.



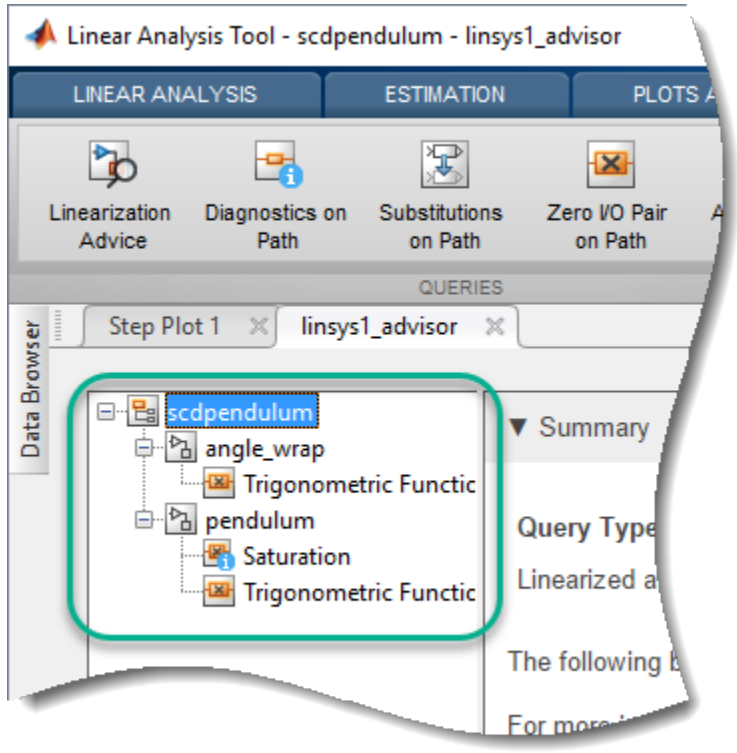
When you select this option and linearize your model, the software opens an **Advisor** tab for troubleshooting your linearization results.



Tip To make viewing the diagnostic information easier, you can minimize the **Data Browser**.

On the **Advisor** tab, you can gain insight into your model linearization by querying the diagnostic information. To do so, use the built-in queries in the **Queries** section, or create custom queries in the **Manage Queries** section.

When you run a query, the navigation tree lists the linearized blocks in your model that match the query search criteria. The tree structure reflects the model hierarchy.



To view a table of all blocks that match the search criteria, in the navigation tree, click the top-level model name. You can also view all blocks in a subsystem that satisfy the query by clicking the subsystem name. Each entry in the table summarizes the linearization diagnostics for a single block.

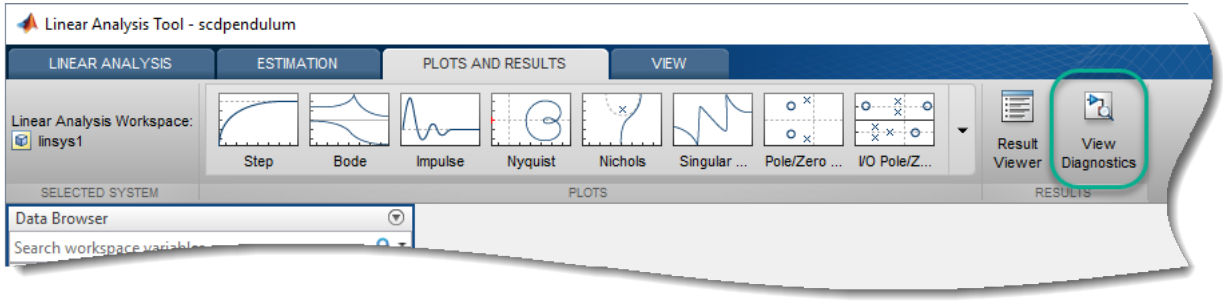
BLOCK PATH	IS ON PATH	CONTRIBUTES TO LINEARIZATION	HAS DIAGNOSTICS	LINEARIZATION METHOD	
scdpendulum/pendulum/Saturation	Yes	No	Yes	Exact	Block Info
scdpendulum/angle_wrap/Trigonometric Function1	Yes	No	No	Perturbation	Block Info
scdpendulum/pendulum/Trigonometric Function	Yes	No	No	Perturbation	Block Info

To view detailed diagnostic information for a block in a table, in the corresponding row, click **Block Info**. You can troubleshoot the block linearization using the detailed diagnostic information. For more information, see "Block Linearization Troubleshooting" on page 4-58.

For an example of interactive troubleshooting using the Linearization Advisor, see "Troubleshoot Linearization Results in Linear Analysis Tool" on page 4-21.

Tip If you close the **Advisor** tab for a given linearization, you can reopen it from the **Plots and Results** tab.

In the **Linear Analysis Workspace**, select the linearized model you want to troubleshoot. Then, click **View Diagnostics**. This option is only available if you enabled the Linearization Advisor before linearizing the model.



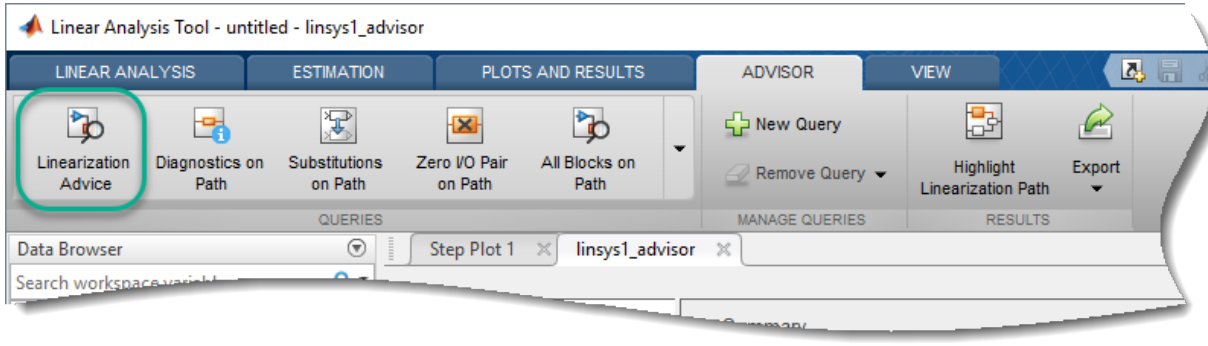
You can also create a `LinearizationAdvisor` object when you linearize models at the command line. You can then troubleshoot the linearization results using the `advise` and `find` functions. For an example, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Blocks That Are Potentially Problematic for Linearization

As a starting point for troubleshooting, the Linearization Advisor searches the linearization diagnostic information for blocks that can cause common linearization issues. These potentially problematic blocks are on the linearization path and satisfy at least one of the following criteria.

Criteria	Description
Blocks with linearization diagnostic messages	Diagnostic messages indicate blocks with configurations or linearizations that correspond to common linearization problems.
Blocks that linearize to zero	Blocks with zero linearizations do not contribute to the linearization result and can remove other blocks from the linearization result.
Blocks with substituted linearizations	Errors in defining substitute linearizations can be difficult to diagnose.

In the Linear Analysis Tool, the diagnostic information for these blocks is listed on the **Advisor** tab when the tab first opens. Also, to access this diagnostic information at any time, in the **Queries** section, click **Linearization Advice**.



You can troubleshoot the linearizations of these blocks using the Linearization Advisor. For more information on troubleshooting block linearizations using diagnostic information, see “Block Linearization Troubleshooting” on page 4-58.

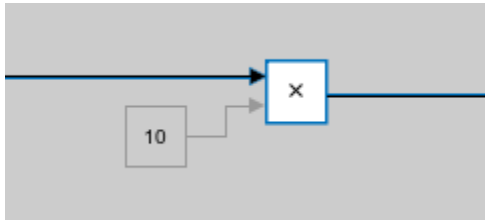
At the command line, the `advise` function returns diagnostic information for these blocks.

Linearization Path

The linearization path is the graphical connection in the Simulink model from the linearization inputs to the linearization outputs. A block is on the linearization path if at least one linearization input is connected to at least one linearization output through that block. For more information on specifying linearization inputs and outputs, see “Specify Portion of Model to Linearize” on page 2-13.

When a block is on the linearization path, its linearization can contribute to the overall model linearization. Blocks that linearize to zero do not contribute to the model linearization and can prevent branches of the linearization path from contributing to the model linearization.

Blocks that are not on the linearization path can still affect the linearization of other blocks, and therefore the model linearization, by modifying the operating point or parameters of the other block. For example, consider the following Product block that is on the linearization path (highlighted in blue):



The constant block is not on the linearization path. However, the value of the constant affects the operating point of the Product block, which in turn affects the linearization from the first input of the Product block to the output.

To visualize the linearization path and view blocks that contribute to the model linearization, you can highlight the linearization path in the Simulink model. For more information, see “Highlight Linearization Path” on page 4-15.

Blocks with Linearization Diagnostic Messages

Linearization diagnostic messages indicate blocks with properties or linearizations that correspond to common linearization problems. Fixing linearization issues identified in diagnostic messages is a good first step when troubleshooting your linearization.

Some block configurations that can generate diagnostic messages include:

- Blocks with nondouble input or output signals, and no predefined exact linearization. Such blocks linearize to zero and generate diagnostic messages.
- Discontinuous blocks linearized at an operating point near a discontinuity. If such blocks are not treated as a gain during linearization, the software can generate diagnostic messages regarding their linearization.
- Blocks with least one input/output pair that linearizes to zero which causes a zero input/output pair in the overall model linearization.
- Blocks that do not support linearization because they do not have a predefined exact linearization and do not support numerical perturbation.

Some diagnostic messages propose solutions to their corresponding linearization issues. For example, when an input signal is outside the saturation limits of a Saturation block, the diagnostic message proposes treating the block as a gain during linearization.

Blocks That Linearize to Zero

A common cause of linearization issues is a block that unexpectedly linearizes to a gain of zero. To diagnose the cause of a zero block linearization, you can consider:

- Any corresponding diagnostic messages. These messages can highlight common causes of zero linearizations and propose potential solutions.
- The block operating point; that is the values of the block states and inputs at the model operating point used for linearization. For example, if the input to a saturation block is outside the block saturation limits, and the block is not configured to linearize as a gain, the block linearizes to zero.
- The block parameters. For example, if a block is configured to use nondouble inputs or states and is linearized using numerical perturbation, it linearizes to zero.

A zero block linearization does not necessarily indicate a linearization problem; that is, you may expect a block to linearize to zero under the expected operating conditions of the model. For example, if a Trigonometric Fcn block is configured as a `sin` function and the input value is $\pi/2$ at the model operating point, then the block linearizes to zero.

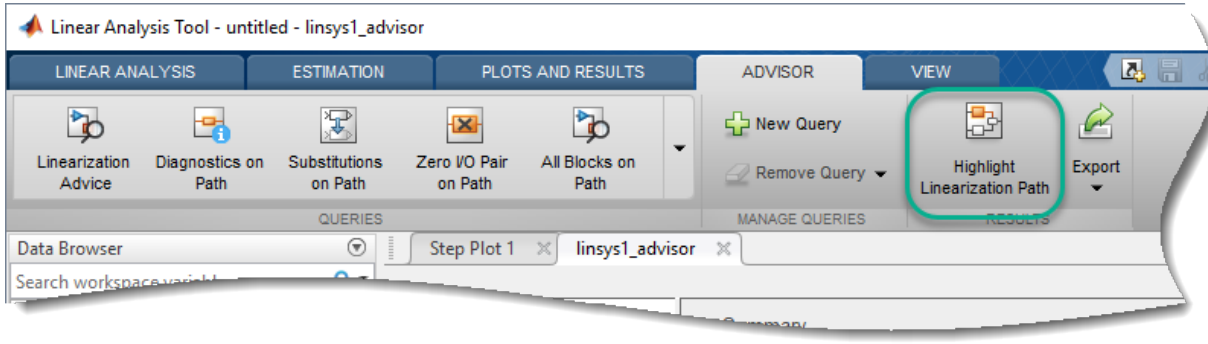
Blocks with Substituted Linearizations

Errors in defining a custom block linearization can be difficult to diagnose. After fixing issues related to diagnostic messages and zero linearizations, if your model still does not linearize as expected, verify that any substituted block linearizations in your model are correct.

For more information on specifying substitute block linearizations, see “When to Specify Individual Block Linearization” on page 2-157.

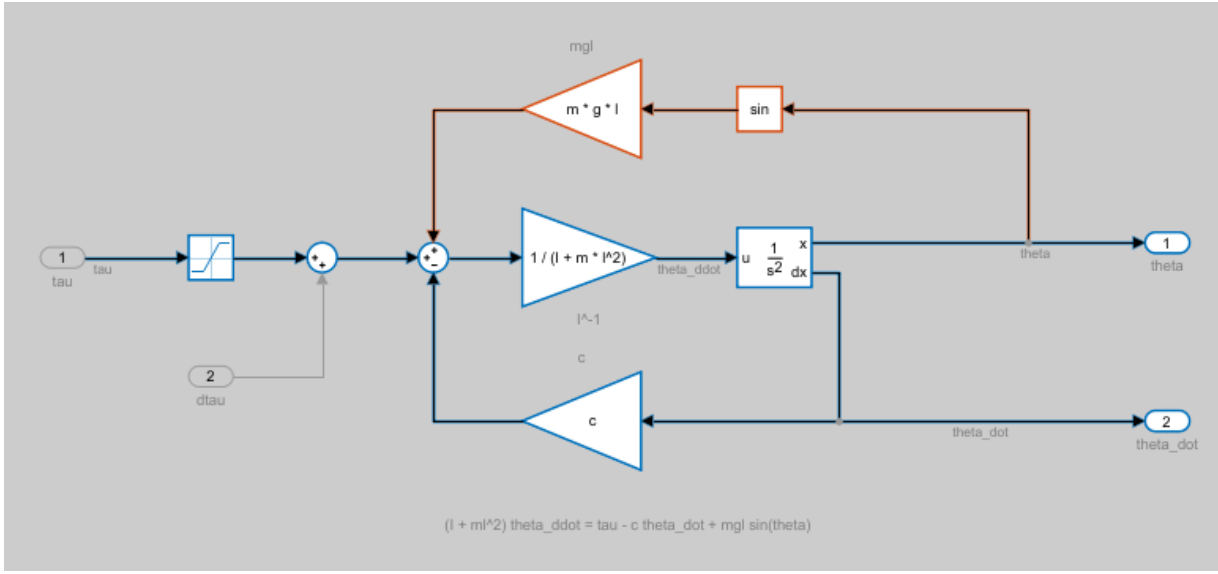
Highlight Linearization Path

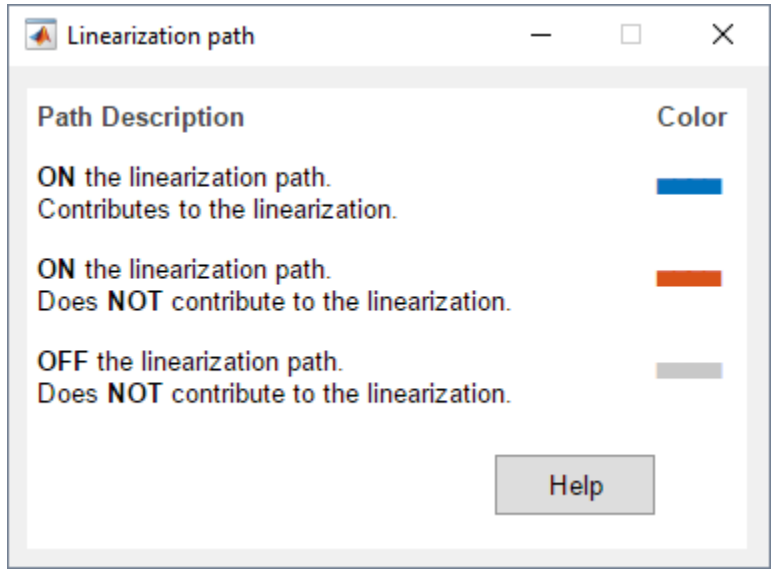
To visualize the linearization path and view blocks that contribute to the model linearization, you can highlight the linearization path in the Simulink model using the Linearization Advisor. After you linearize your model with the Linearization Advisor enabled, to highlight the linearization path, in the Linear Analysis Tool, on the **Advisor** tab, click **Highlight Linearization Path**.



The software highlights the linearization path in the model, showing which blocks are on the path and which blocks contribute to the model linearization. Blocks highlighted in:

- Blue are on the linearization path and contribute to the model linearization.
- Red are on the linearization path, but do not contribute to the model linearization.
- Gray are not on the linearization path and do not contribute to the model linearization.





To turn off the highlighting, close the Linearization path dialog box.

You can also highlight the linearization path from the command line using the `highlight` function.

Find Specific Blocks in Linearization Results

If your model still does not linearize as you expect after fixing linearization issues related to potentially problematic blocks, you can query the Linearization Advisor for additional block diagnostic information. You can gain insight into your model linearization using this information. For example, you can investigate:

- Blocks that are linearized using numerical perturbation.
- Sampling rates of block linearizations in multirate models by finding blocks with a specified sample time.
- Blocks that have delays that can cause linearizations issues.
- Blocks that are not on the linearization path.

For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

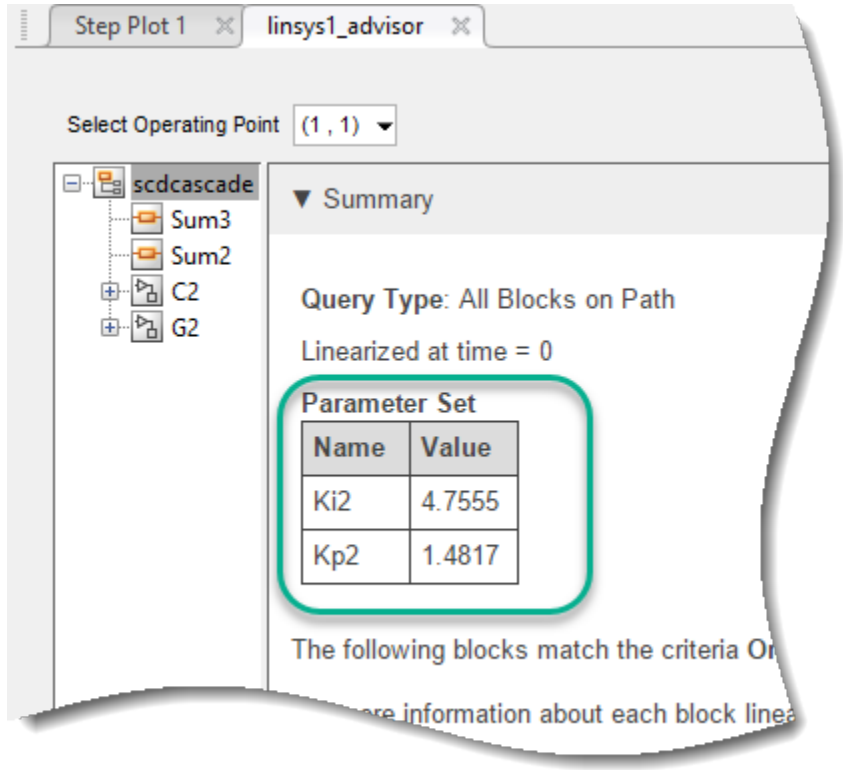
Troubleshoot Batch Linearizations

If you linearize your model at multiple operating points, you can troubleshoot each resulting linear model using Linearization Advisor.

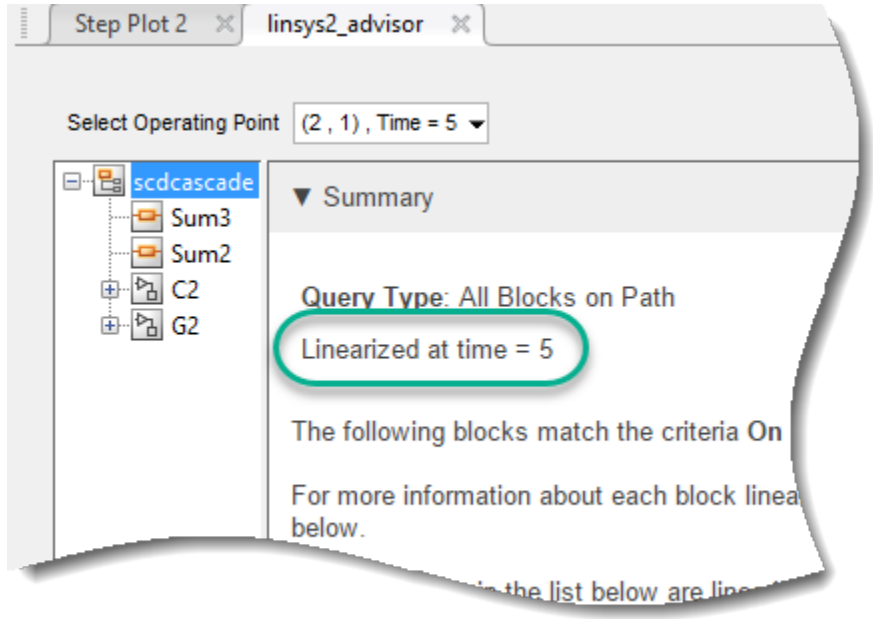
After batch linearizing the model, on the **Advisor** tab, in the **Select Operating Point** drop-down list, select the operating point for which you want to troubleshoot the linearization.

If you batch linearized your model using:

- Parameter variation, the linearization summary shows the parameter values that correspond to the selected operating point.



- Multiple simulation snapshot times, the linearization summary shows the time at which the model was linearized.



- Multiple trimmed operating points, the linearization summary does not show additional information about the operating point. To view details about the operating points, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select the operating point array used for linearization. In the same drop-down list, select `Edit`.

  Then, in the Edit dialog box, in the **Select Operating Point** drop-down list, select an operating point. The location of the operating point in this drop-down list corresponds to the location in the drop-down list on the **Advisor** tab.

See Also

Apps

Linear Analysis Tool

Functions

`advise`

More About

- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
- “Troubleshoot Linearization Results in Linear Analysis Tool” on page 4-21
- “Troubleshoot Linearization Results at Command Line” on page 4-40

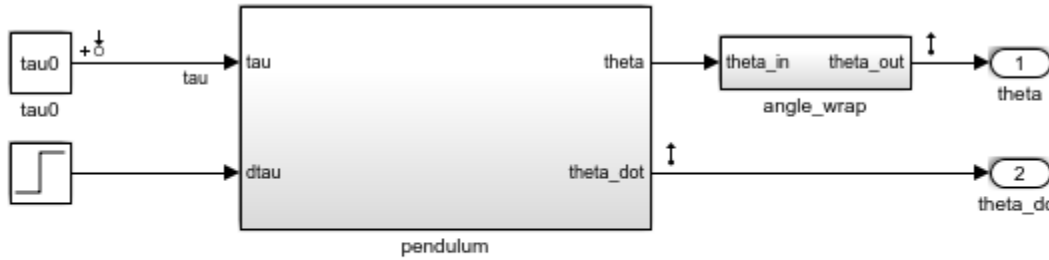
Troubleshoot Linearization Results in Linear Analysis Tool

This example shows how to use the Linearization Advisor to debug the linearization of a pendulum model in the Linear Analysis Tool.

Setup Model

Open the Simulink model.

```
mdl = 'scdpendulum';
open_system(mdl)
```



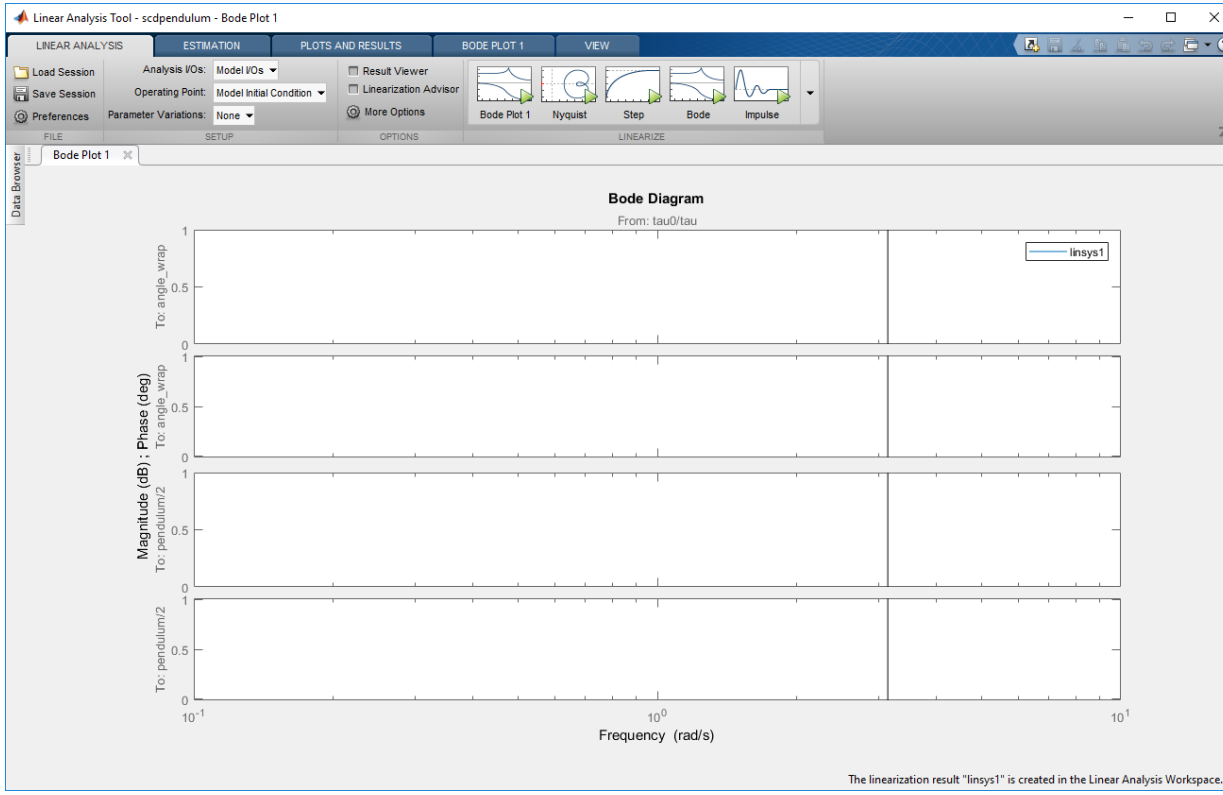
The initial condition for the pendulum angle is 90 degrees counterclockwise from the upright unstable equilibrium of 0 degrees. The initial condition for the pendulum angular velocity is 0 deg/s. The nominal torque to maintain this state is -49.05 N m. This configuration is saved as the model initial condition.

Open Linear Analysis Tool and Linearize Model

To open the Linear Analysis Tool, in the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

To linearize the model at the model initial condition, in the Linear Analysis Tool, on the **Linear Analysis** tab, click **Bode**.

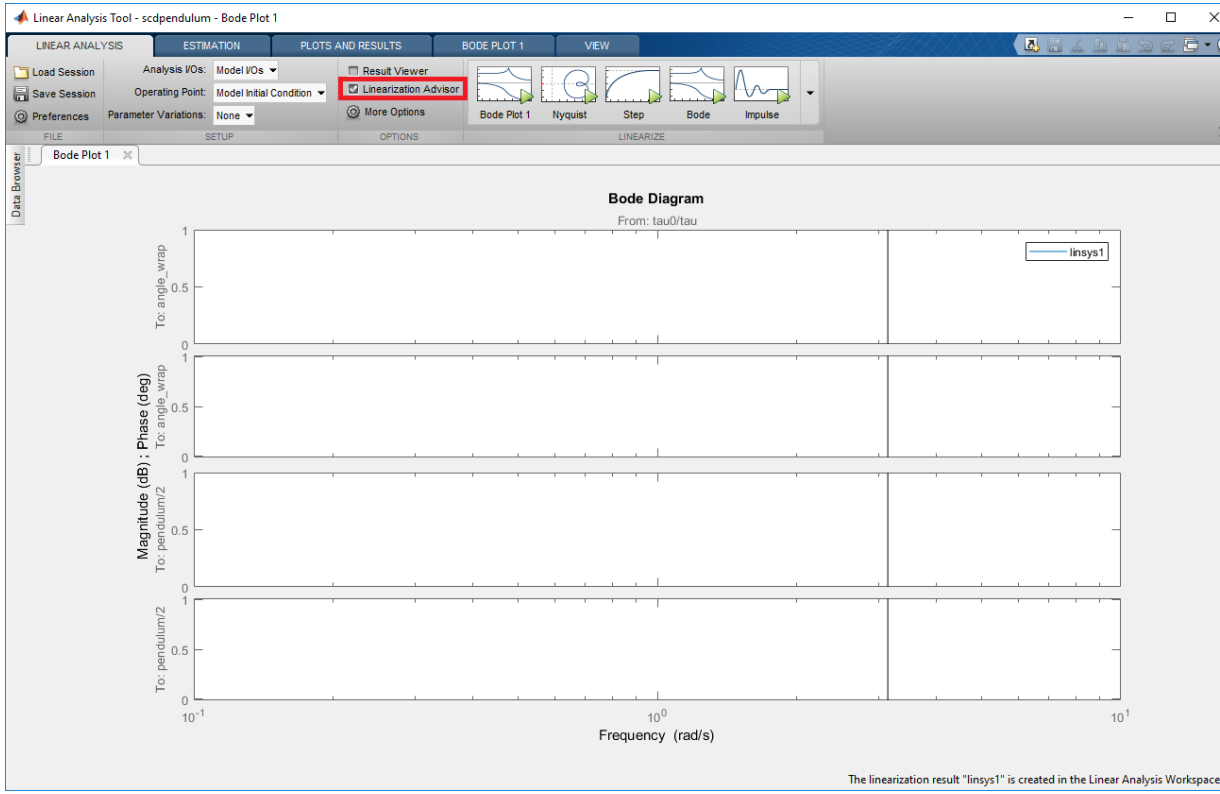
The software linearizes the model and plots its frequency response.



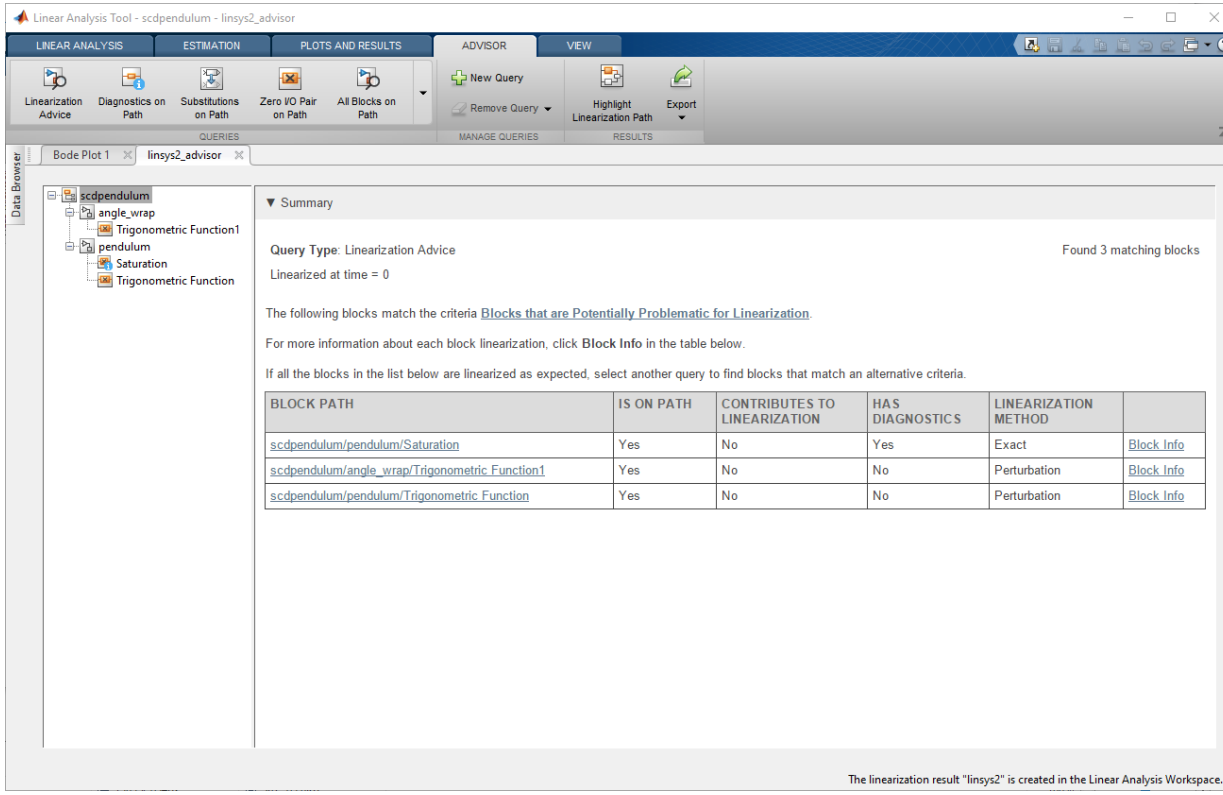
As can be seen from the Bode plot, the system has linearized to zero such that the torque has no effect on the angle or angular velocity. You can explore why this is the case using the Linearization Advisor.

Linearize Model with Advisor Enabled

To relinearize the model and generate an advisor, select **Linearization Advisor**, and click **Bode Plot 1**.



The software linearizes the model, creates the **linsys2_advisor** document, and opens the **Advisor** tab.

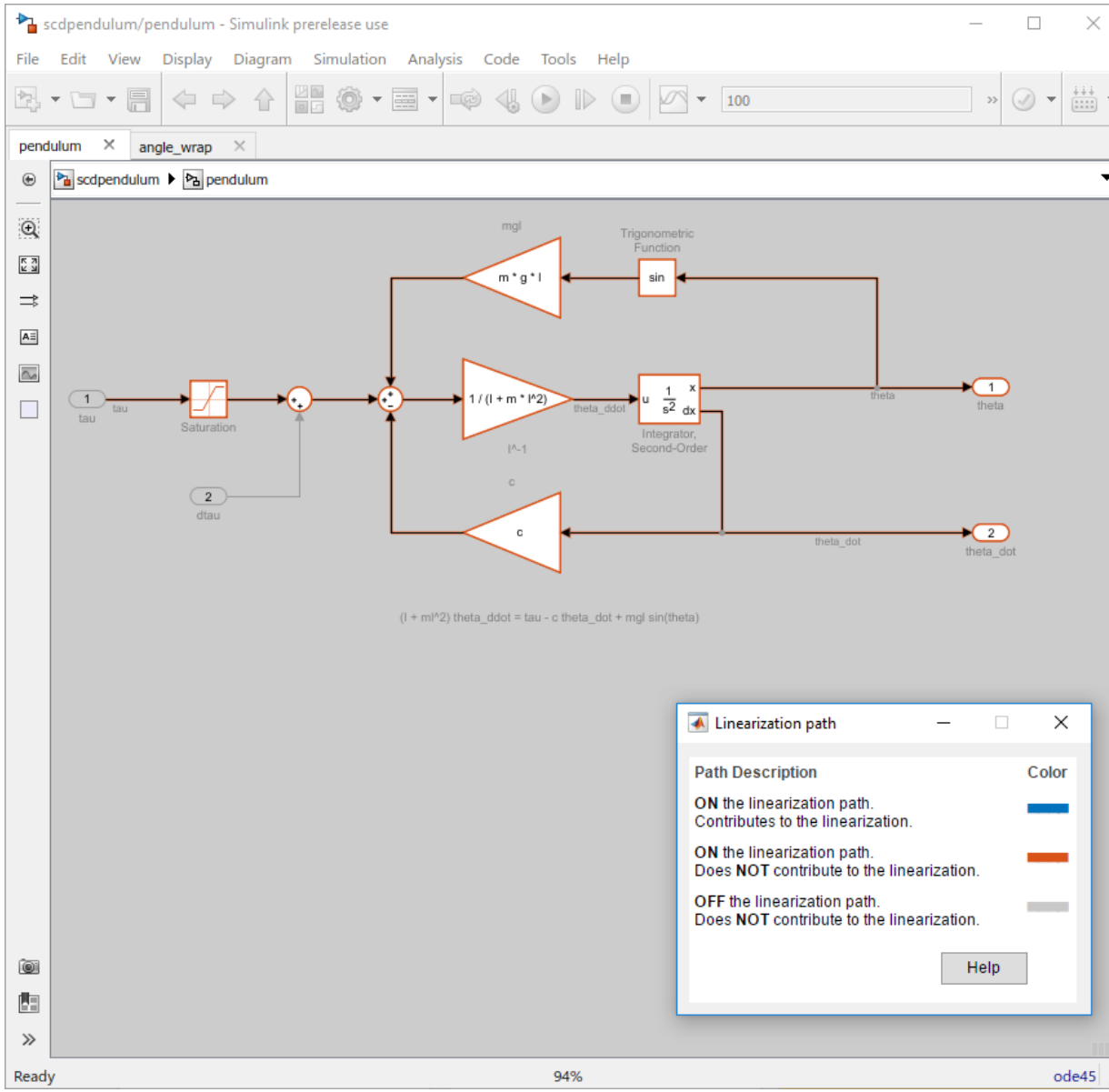


Highlight Linearization Path

To show the linearization path for the current linearization, on the **Advisor** tab, click **Highlight Linearization Path**. In the Linearization path dialog box, the blocks highlighted in:

- Blue numerically influence the model linearization.
- Red are on the linearization path but do not influence the model linearization for the current operating point and block parameters.

For convenience, only the blocks underneath the pendulum subsystem are shown.

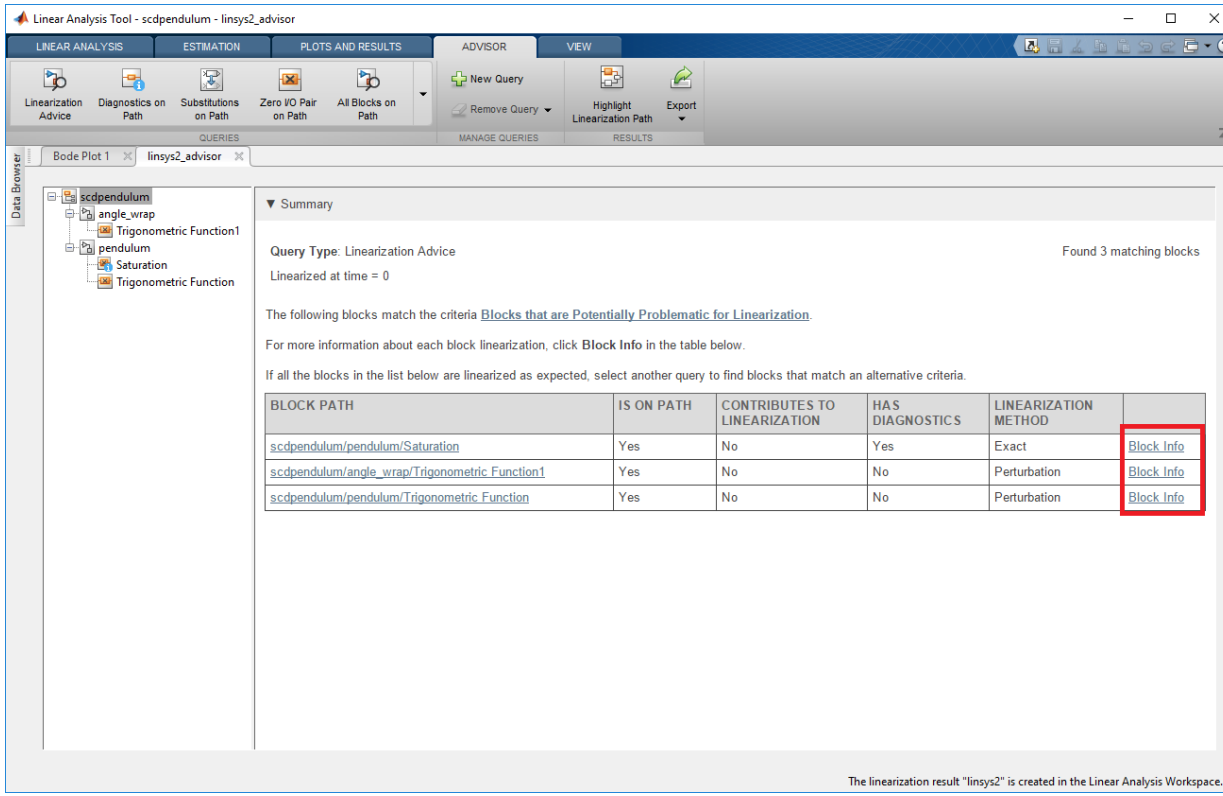

scdpendulum/pendulum - Simulink prerelease use
File Edit View Display Diagram Simulation Analysis Code Tools Help
100
pendulum
angle_wrap
scdpendulum
pendulum
mgl
m * g * l
Trigonometric Function
sin
1
tau
tau
Saturation
1 / (I + m * l^2)
theta_ddot
I^-1
u
1
s2
x
dx
Integrator, Second-Order
theta
1
theta
2
dtau
c
c
theta_dot
2
theta_dot
(I + ml^2) theta_ddot = tau - c theta_dot + mgl sin(theta)
Linearization path
Path Description
Color
ON the linearization path.
Contributes to the linearization.
ON the linearization path.
Does NOT contribute to the linearization.
OFF the linearization path.
Does NOT contribute to the linearization.
Help
Ready
94%
ode45

In this case, since the model linearized to zero, there are no blocks that contribute to the linearization.

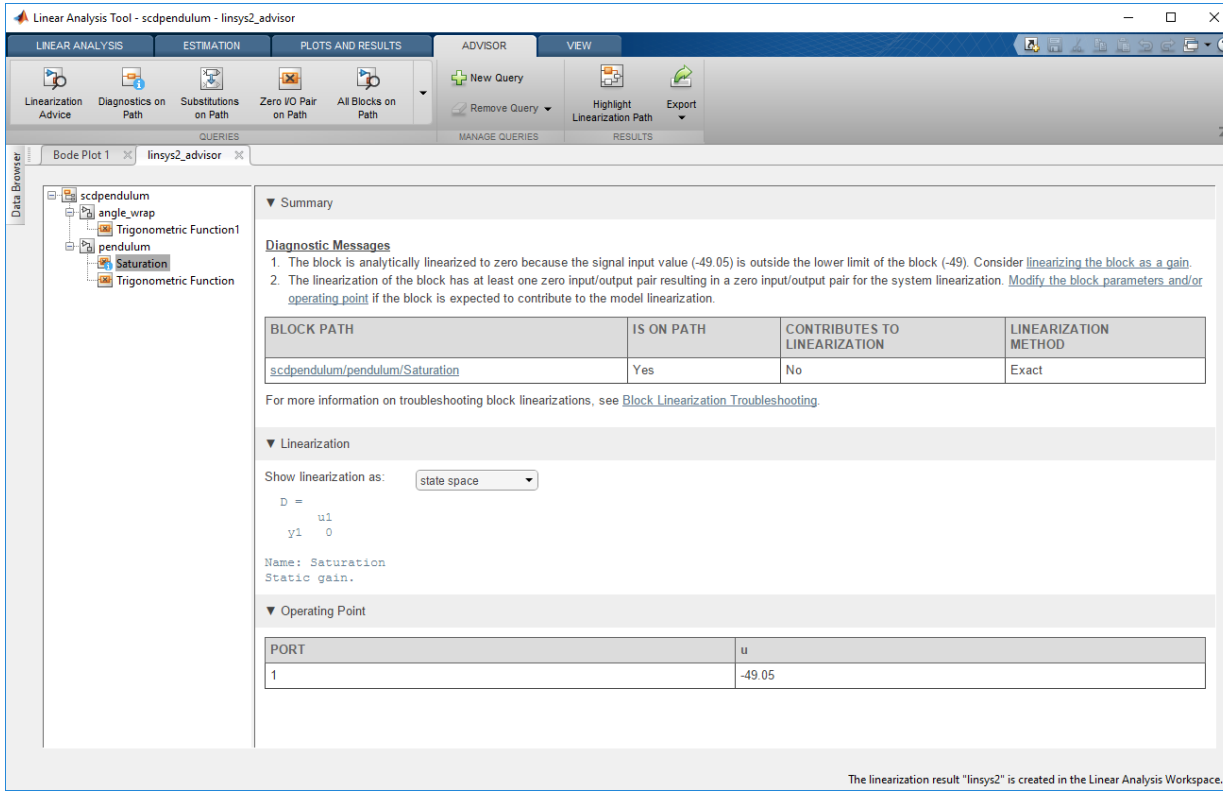
Investigate Potentially Problematic Blocks Using Advisor

The **linsys2_advisor** document shows a table listing blocks that may be problematic for the linearization.

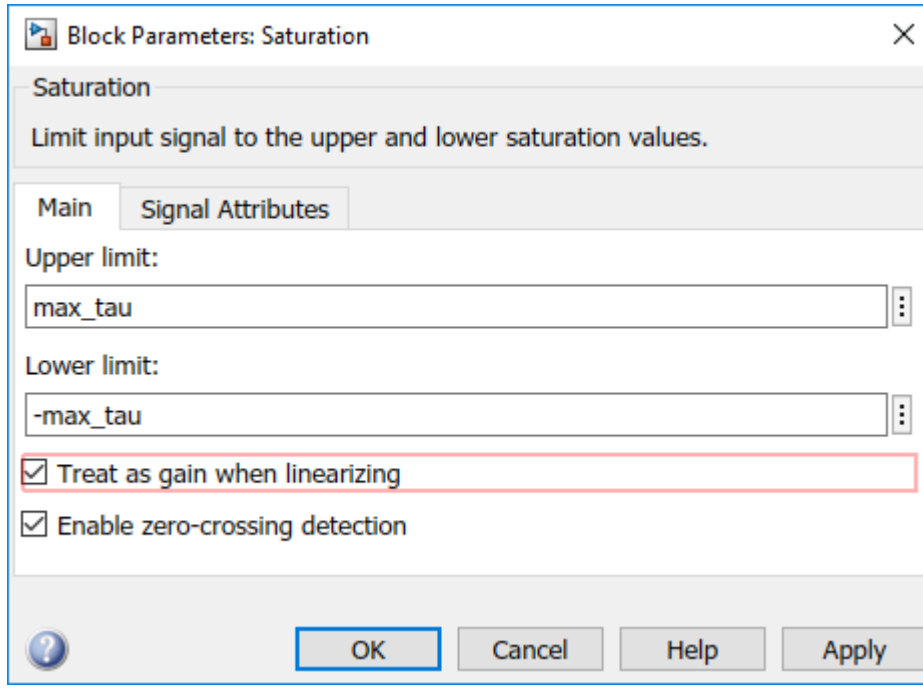
To view more information about a specific block linearization, in the corresponding row of the table, click **Block Info**.



In this case, three blocks are reported by the advisor, a Saturation block and two Trigonometric Function blocks. Investigate the Saturation block first since it has diagnostics. To do so, in the first row of the table, click **Block Info**.



There are two diagnostic messages for the Saturation block. The first message indicates that the block is linearized outside of its lower saturation limit of `-49`, since the input operating point is `-49.05`. The message also states the block can be linearized as a gain, which will linearize the block as `1` regardless of the input operating point. To do so, first click **linearizing the block as a gain**, which highlights the corresponding parameter in the block dialog box. Then, select the **Treat as gain when linearizing** parameter.

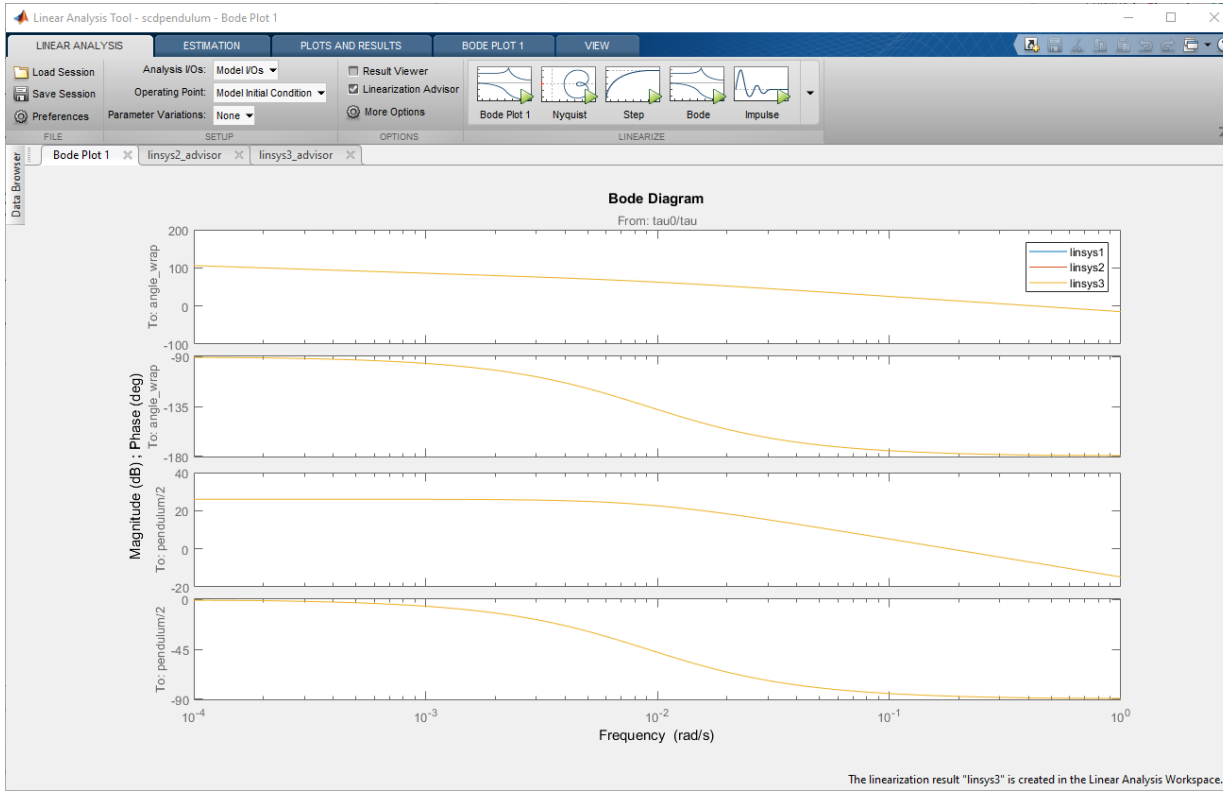


The second message states that the linearization of this block causes the model to linearize to zero. As shown in the **Linearization** section, the block is linearized to zero. Therefore, modifying the block linearization is a good first step toward obtaining a nonzero model linearization.

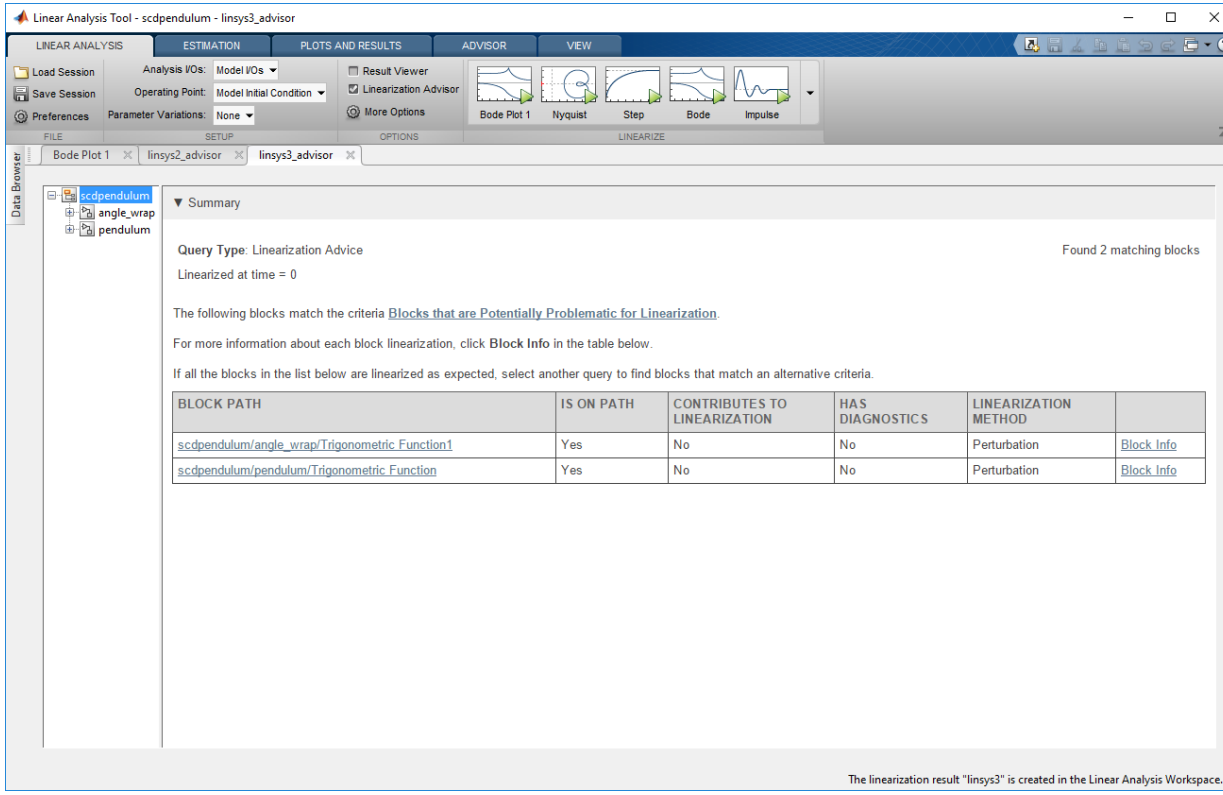
Relinearize Model

After setting the Saturation block to be treated as a gain, relinearize the model. For now, ignore the diagnostics for the two Trigonometric Function blocks.

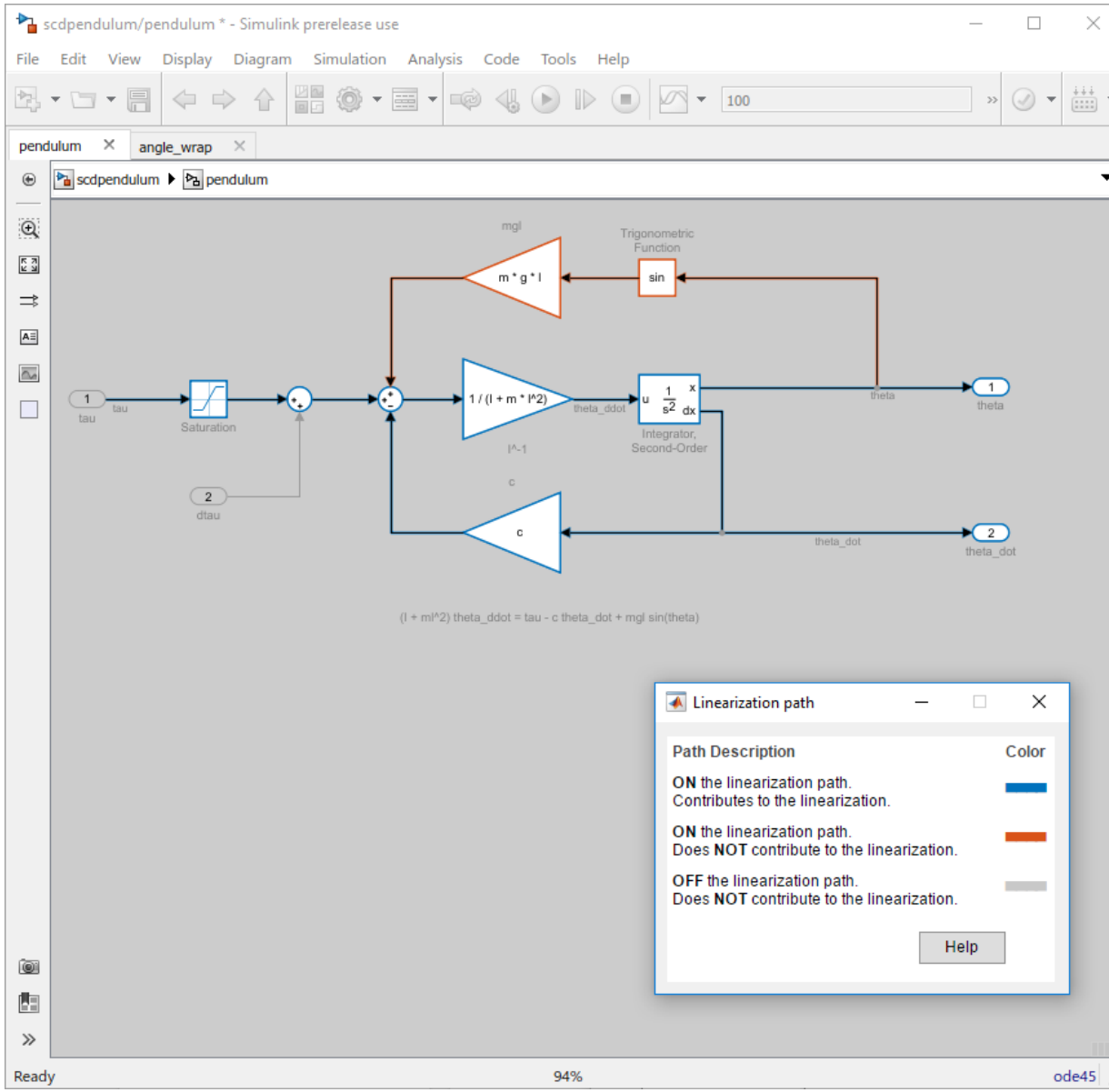
To relinearize the model, on the **Linear Analysis** tab, click **Bode Plot 1**. The **Bode Plot 1** document updates, showing the nonzero response of `linsys3`.



In the corresponding **linsys_advisor3** document, the Saturation block is no longer listed. However, the two Trigonometric Function blocks are still shown.

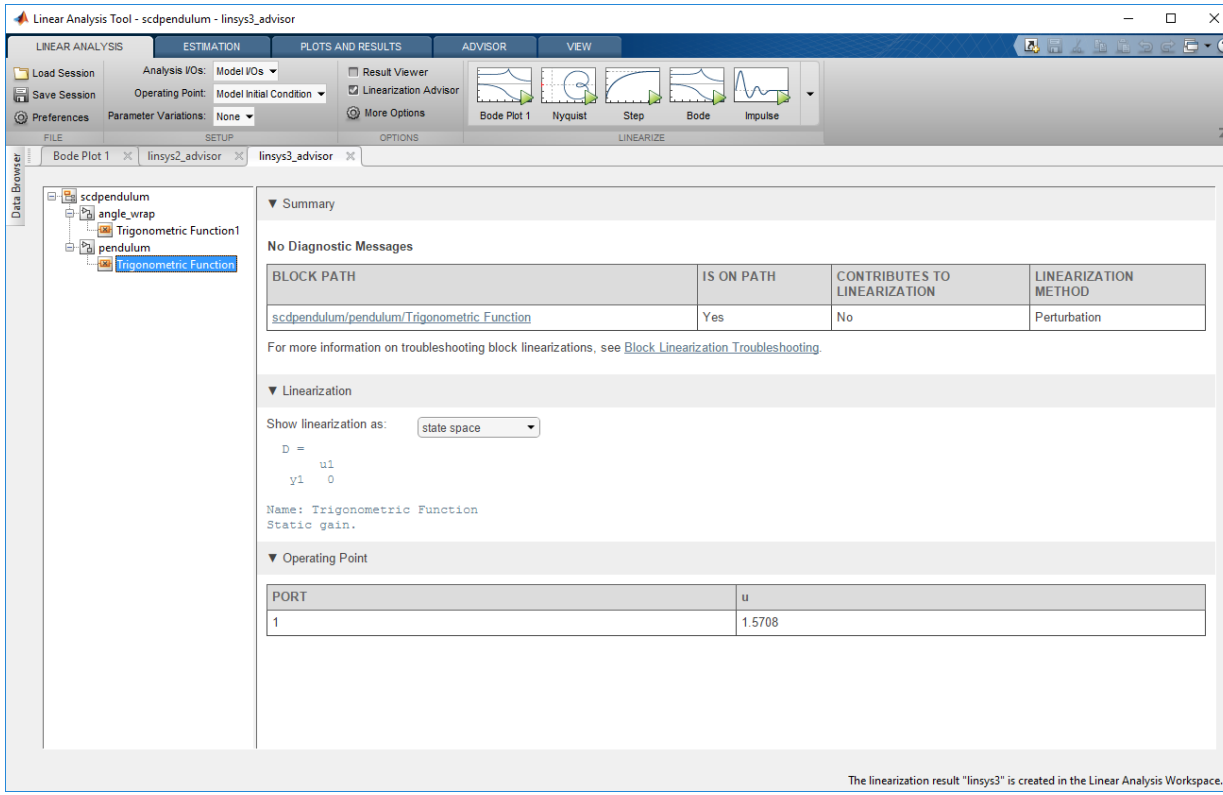


Highlight the linearization path.


scdpendulum/pendulum * - Simulink prerelease use
File Edit View Display Diagram Simulation Analysis Code Tools Help
100
pendulum
angle_wrap
scdpendulum
pendulum
mgl
m * g * l
Trigonometric Function
sin
1
tau
tau
Saturation
2
dtau
1 / (I + m * l^2)
I^-1
theta_ddot
u
1
s2
x
dx
Integrator, Second-Order
theta
1
theta
c
c
theta_dot
2
theta_dot
(I + ml^2) theta_ddot = tau - c theta_dot + mgl sin(theta)
Linearization path
Path Description
Color
ON the linearization path.
Contributes to the linearization.
ON the linearization path.
Does NOT contribute to the linearization.
OFF the linearization path.
Does NOT contribute to the linearization.
Help
Ready
94%
ode45

Most of the blocks are now contributing to the model linearization, except for the paths going through the listed Trigonometric Function blocks.

To understand why these blocks are not contributing to the linearization, navigate to the blocks from the **linsys3_advisor** document. For example, click **Block Info** in the second row of the table.



For this Trigonometric Function block, the linearization is zero and the input operating point is $\pi/2 = 1.5708$.

You can find the linearization of the block analytically by taking the first derivative of the `sin` function with respect to the inputs:

$$\frac{\partial}{\partial u}\sin(u) = \cos(u)$$

Therefore, when evaluated at $u = \pi/2$ the linearization of the block is zero. The source of the input is the first output of the second-order integrator, which is dependent upon the state **theta**. Therefore, this block will linearize to zero if $\theta = \pi/2 + k\pi$, where k is an integer. The same condition applies for the other Trigonometric Function in the angle_wrap subsystem.

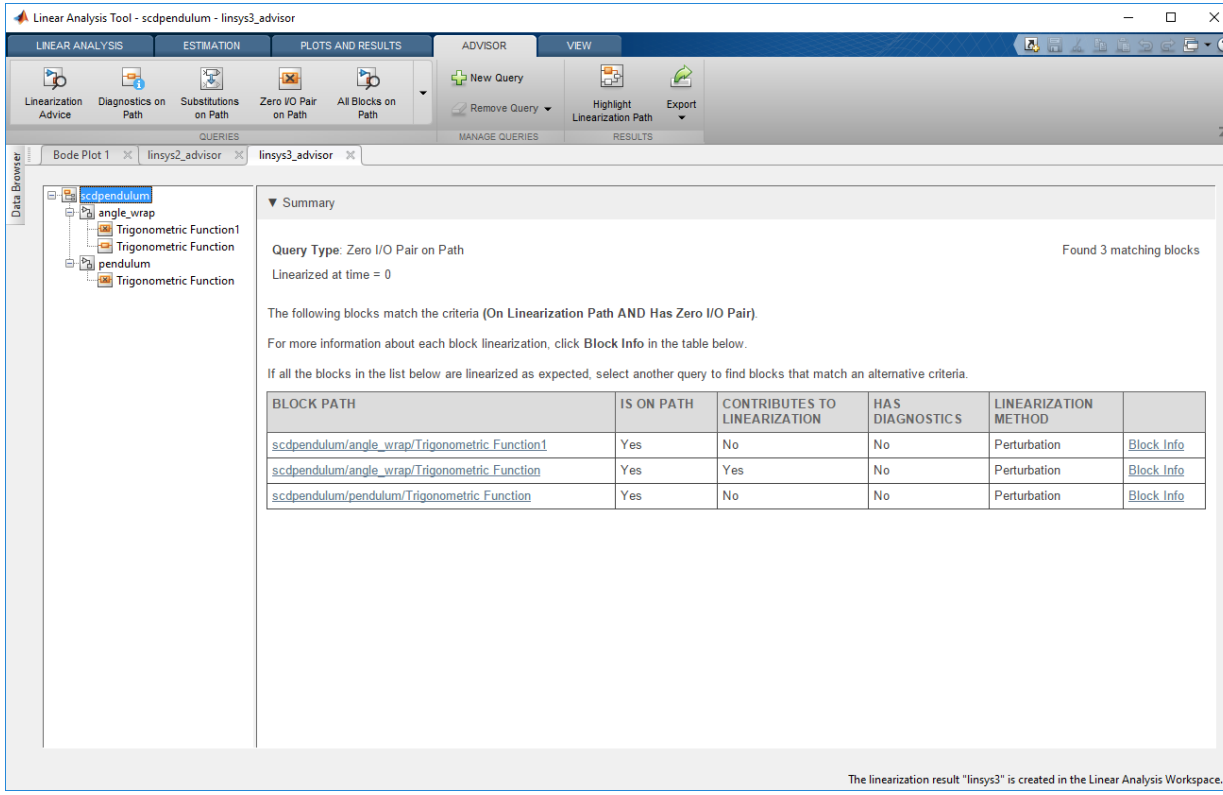
If these blocks are not expected to linearize to zero, you can modify the operating point state **theta**, and relinearize the model.

Run Prebuilt Advisor Queries

The Linearization Advisor provides a set of prebuilt queries for filtering block diagnostics. For example, the **Linearization Advice** query is the default query run when the advisor is first created and includes blocks on the path that:

- Have diagnostic messages regarding the block linearization.
- Linearized to zero.
- Have substituted linearizations.

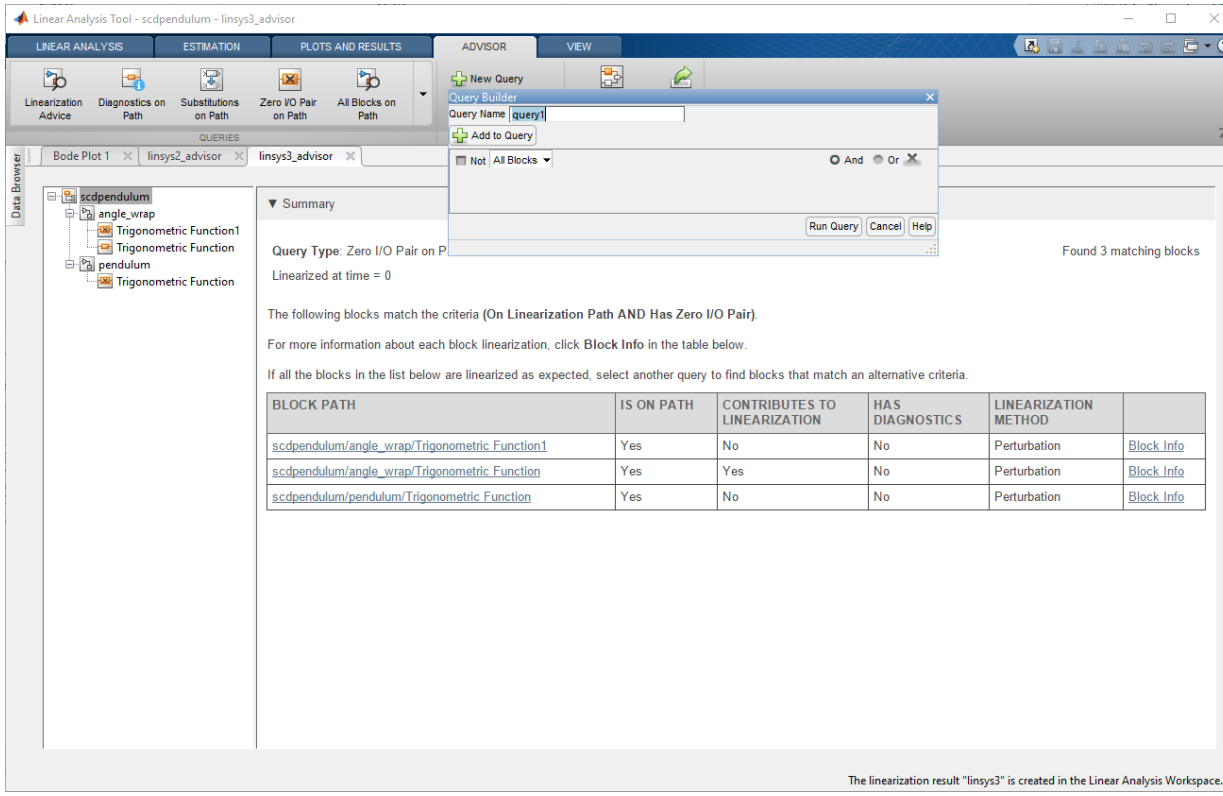
To run a different prebuilt query, on the **Advisor** tab, in the **Queries** gallery, click the query. For example, click **Zero I/O Pair on Path**.



This query returns blocks with linearizations that have output channels that cannot be reached by any input channel, or input channels that have no influence on any output channels. For example, the second block in the table is a Trigonometric Function block configured as `atan2`. The first input of this block cannot reach the only output.

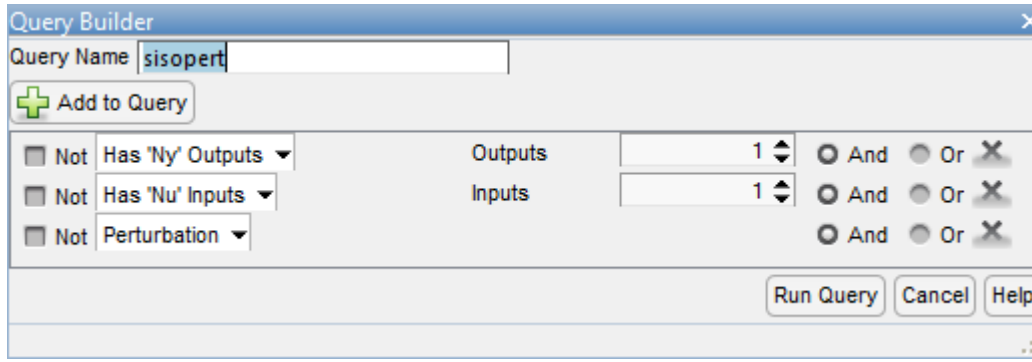
Create and Run Custom Queries

The Linearization Advisor also provides a Query Builder for creating custom queries. You can use these queries to find blocks in your model that match specific criteria. For example, to find all SISO blocks that are numerically perturbed, first open the Query Builder. To do so, on the **Advisor** tab, click **New Query**.



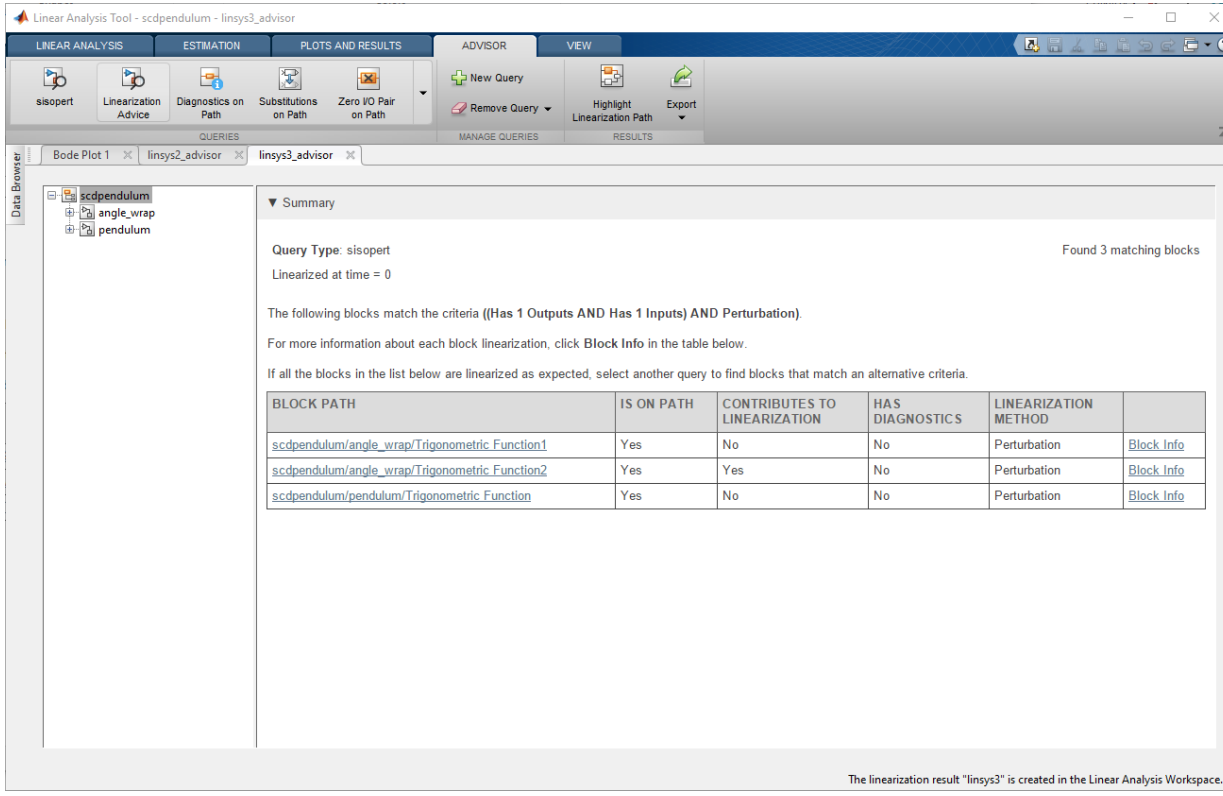
In the Query Builder dialog box:

1. Specify the **Query Name** as `sisopert`.
2. In the drop-down list, select `Has 'Ny' Outputs'`, and specify `1` in the **Outputs** box.
3. To add another component to the query, click **Add to Query**.
4. In the second drop-down list, select `Has 'Nu' Inputs'`, and specify `1` in the **Inputs** box.
5. Click **Add to Query**.
6. In the third drop-down list, select `Perturbation`.

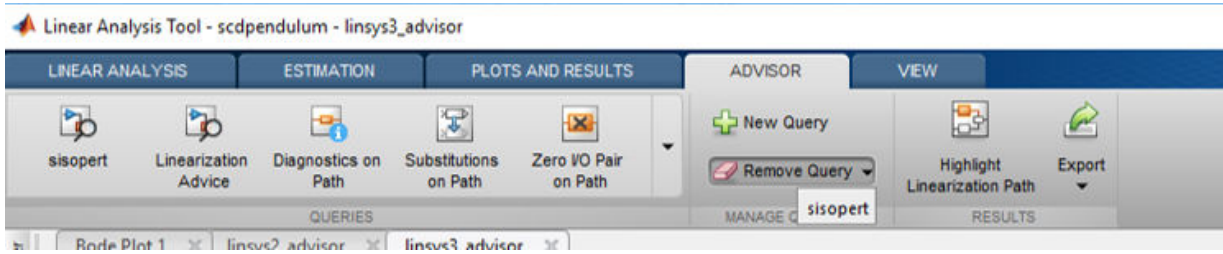


Click **Run Query**.

The **linsys3_advisor** document shows the blocks that match the specified query criteria, and the **sisopert** query is added to the **Queries** gallery.

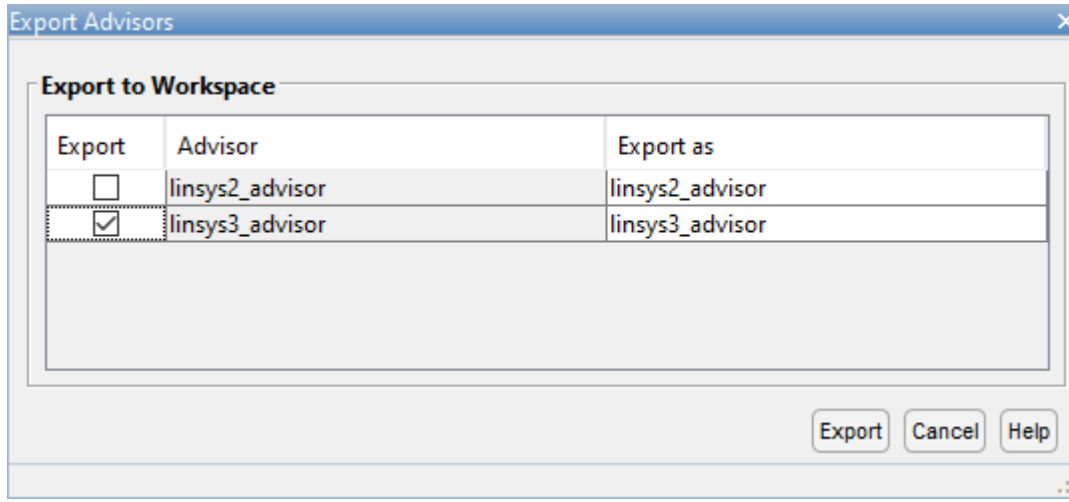


To remove the **sisopert** query, on the Advisor tab, click **Remove Query**, and select **sisopert**.



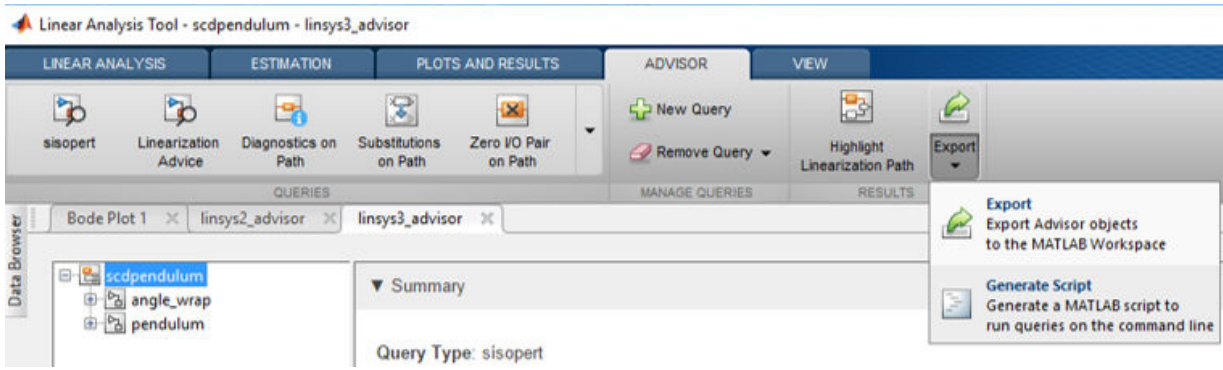
Export Advisor and Generate MATLAB Script

You can also debug model linearizations using the Linearization Advisor command-line functions. To export the advisor object to the MATLAB workspace, click **Export**. Then, in the Export Advisors dialog box, select one or more advisors to export. For example, select **linsys3_advisor**.



Click **Export**.

Alternatively, you can generate a MATLAB script that automates the linearization, extraction of the advisor, generation of custom queries, and running of queries. To generate this script, click the **Export** split button, then select **Generate Script**.



```
bdclose(mdl)
```

See Also

Apps

Linear Analysis Tool

More About

- “Identify and Fix Common Linearization Issues” on page 4-8
- “Troubleshoot Linearization Results at Command Line” on page 4-40
- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52

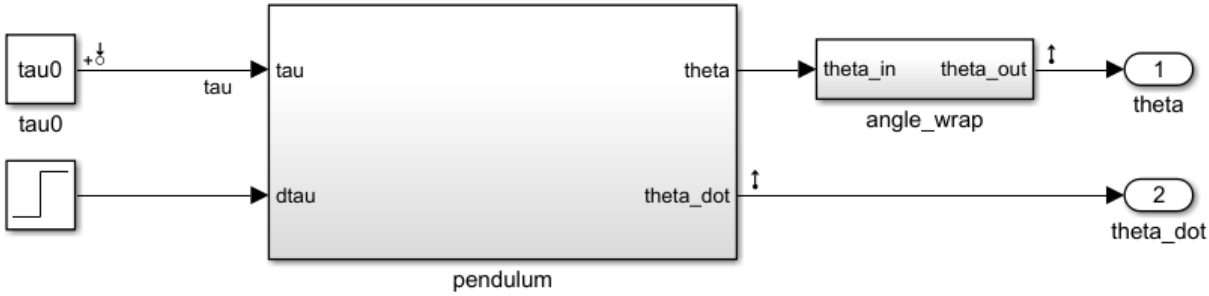
Troubleshoot Linearization Results at Command Line

This example shows how to debug the linearization of a Simulink model at the command line using a `LinearizationAdvisor` object. You can also troubleshoot linearization results interactively. For more information, see "Troubleshoot Linearization Results in Linear Analysis Tool" on page 4-21.

For this example, troubleshoot the linearization of a pendulum model.

Open the model.

```
mdl = 'scdpendulum';
open_system(mdl)
```



The initial condition for the pendulum angle is 90 degrees counterclockwise from the upright unstable equilibrium of 0 degrees. The initial condition for the pendulum angular velocity is 0 deg/s. The nominal torque to maintain this state is -49.05 N m. This configuration is saved as the model initial condition.

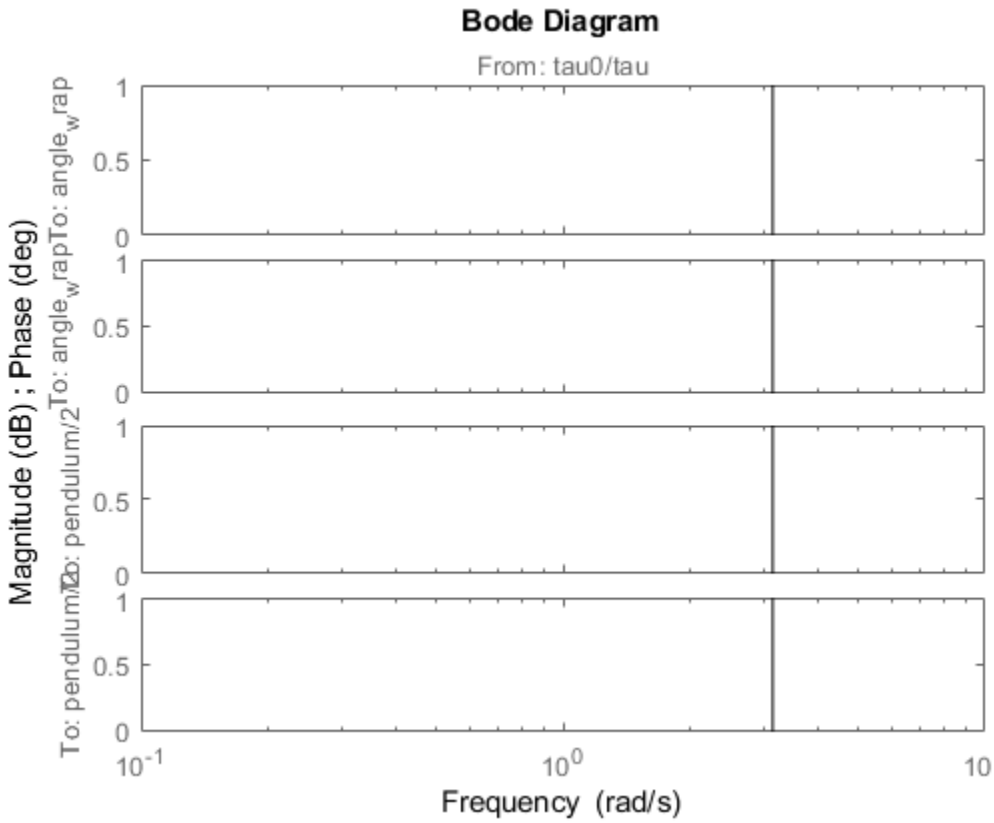
Linearize Model

Linearize the model using the analysis points defined in the model and the model operating point.

```
io = getlinio(mdl);
linsys = linearize(mdl,io);
```

To check the linearization result, plot its Bode response.

```
bode(linsys)
```



The model linearized to zero such that the torque, `tau`, has no effect on the angle or angular velocity. To find the source of the zero linearization, you can use a `LinearizationAdvisor` object.

Linearize Model with Advisor Enabled

To collect diagnostic information during linearization and create an advisor for troubleshooting, first create a `linearizeOptions` option set, specifying the `StoreAdvisor` option as `true`.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Linearize the Simulink model using this option set. Return the `info` output argument, which contains linearization diagnostic information in a `LinearizationAdvisor` object.

```
[linsys1,~,info] = linearize(mdl,io,opt);
```

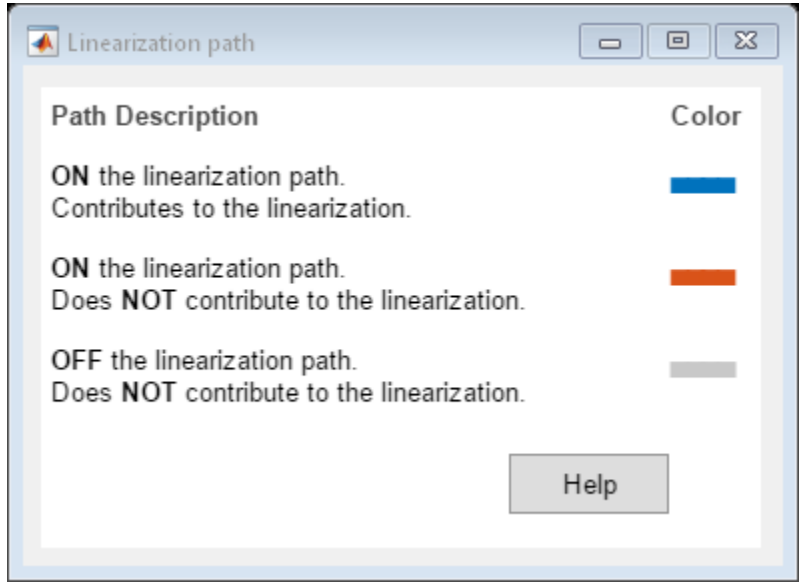
Extract the `LinearizationAdvisor` object.

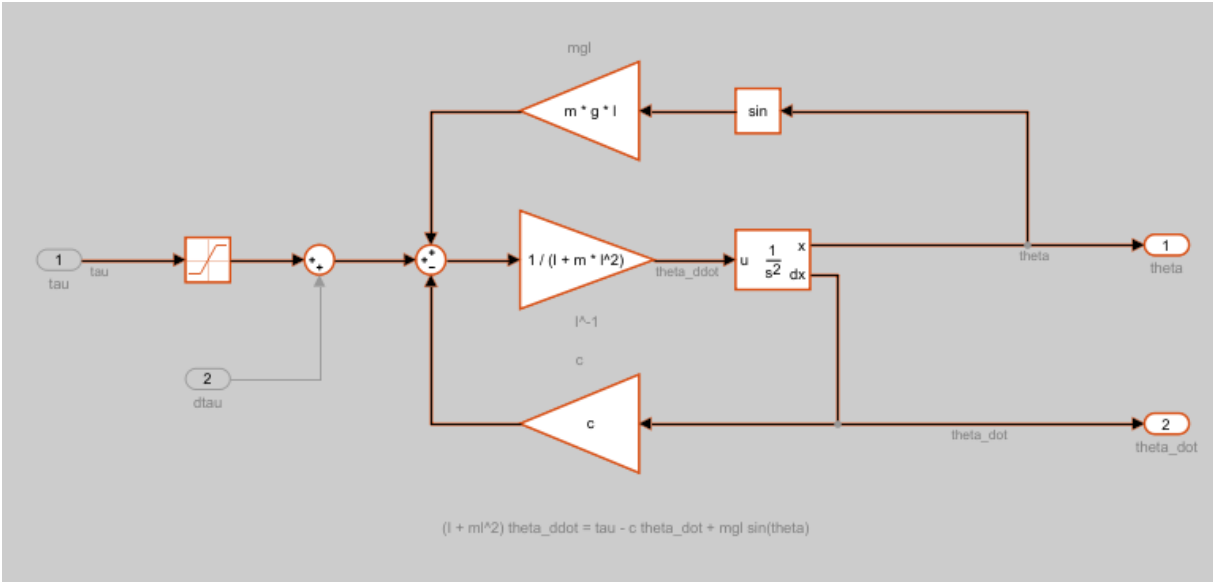
```
advisor = info.Advisor;
```

Highlight Linearization Path

To show the linearization path for the current linearization, use `highlight`. Also, open the `pendulum` subsystem.

```
highlight(advisor)
```





As shown in the Linearization path dialog box, the blocks highlighted in:

- Blue numerically influence the model linearization.
- Red are on the linearization path but do not influence the model linearization for the current operating point and block parameters.

Since the model linearized to zero, there are no blocks that contribute to the linearization.

Investigate Potentially Problematic Blocks

To obtain diagnostic information for blocks that may be problematic for linearization, use `advise`. This function returns a new `LinearizationAdvisor` object that contains information on blocks on the linearization path that satisfy at least one of the following criteria:

- Have diagnostic messages regarding their linearization
- Linearize to zero
- Have substituted linearizations

```
adv1 = advise(advisor);
```

View a summary of the diagnostic information for these blocks, use `getBlockInfo`.

```
getBlockInfo(adv1)
```

```
ans =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

In this case, the advisor reports three potentially problematic blocks, a `Saturation` block and two `Trignonometric Function` blocks. When you run this example in MATLAB, the block paths display as hyperlinks. To go to one of these blocks in the model, click the corresponding block path hyperlink.

To view more information about a specific block linearization, use `getBlockInfo`. For information on the available diagnostics, see `BlockDiagnostic`.

For example, obtain the diagnostic information for the `Saturation` block.

```
diag = getBlockInfo(adv1,1)
```

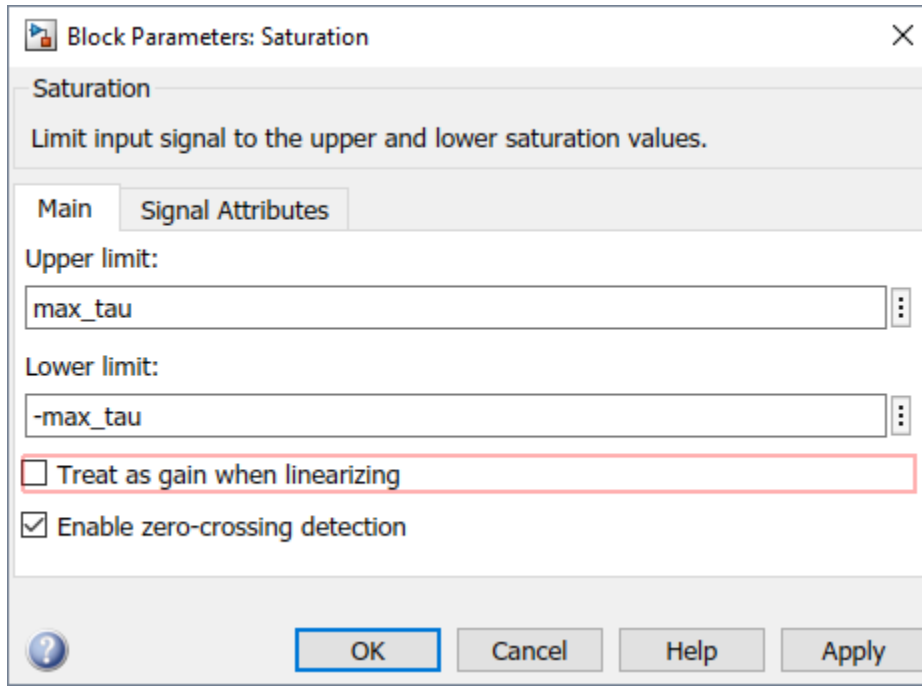
```
diag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

This block has two diagnostic messages regarding its linearization result. The first message indicates that the block is linearized outside of its lower saturation limit of `-49`, since the input operating point is `-49.05`.

The message also indicates that the block can be linearized as a gain, which linearizes the block as `1` regardless of the input operating point.

When you run this example in MATLAB, the text **`linearizing the block as a gain`** displays as a hyperlink. To open the Block Parameters dialog box for the

`Saturation` block, and highlight the option for linearizing the block as a gain, click this hyperlink.



Select **Treat as gain when linearizing**, and click **OK**.

Alternatively, you can set this parameter from the command line.

```
set_param('scdpendulum/pendulum/Saturation','LinearizeAsGain','on')
```

The second diagnostic message states that the linearization of this block causes the overall model to linearize to zero. View the linearization of this block.

```
diag.Linearization
```

```
ans =

  D =
       u1
   y1   0
```

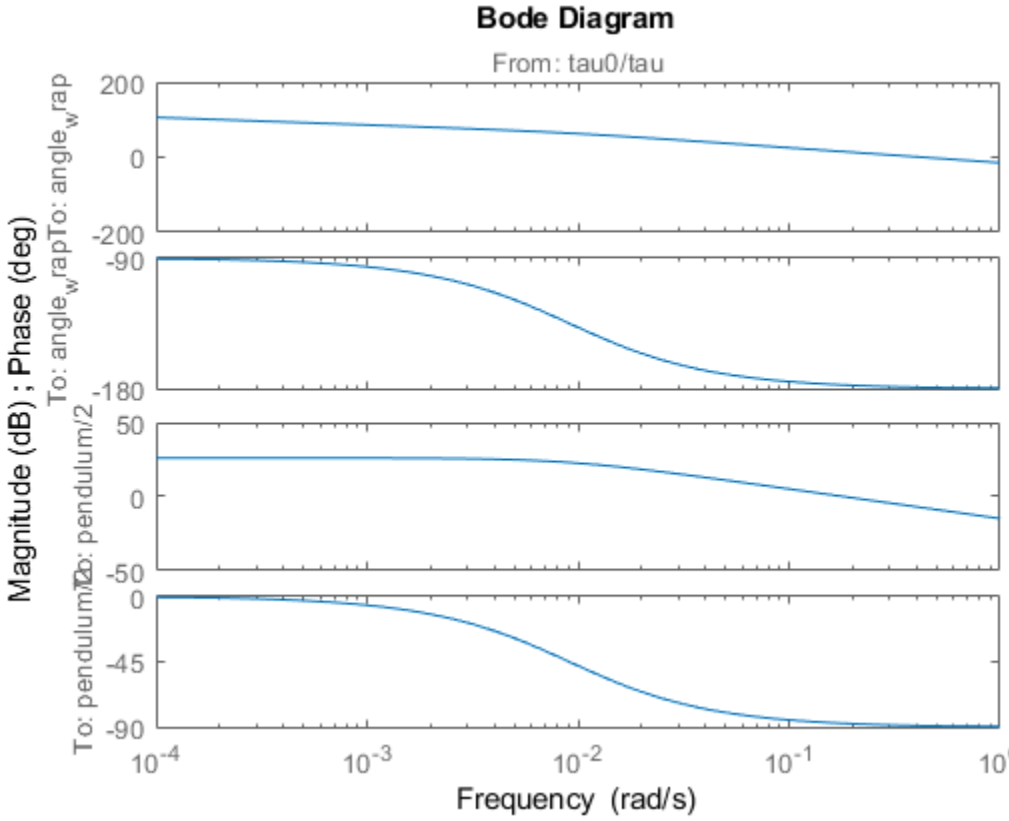
```
Name: Saturation
Static gain.
```

Since this block linearized to zero, modifying the block linearization by treating it as a gain is a good first step toward obtaining a nonzero model linearization.

Relinearize Model

To see the effect of treating the `Saturation` block as a gain, relinearize the model, and plot its Bode response.

```
[linsys2,~,info] = linearize(mdl,io,opt);
bode(linsys2)
```



The model linearization is now nonzero.

To check if any blocks are still potentially problematic for linearization, extract the advisor object, and use the `advise` function.

```
advisor2 = info.Advisor;
adv2 = advise(advisor2);
```

View the block diagnostic information.

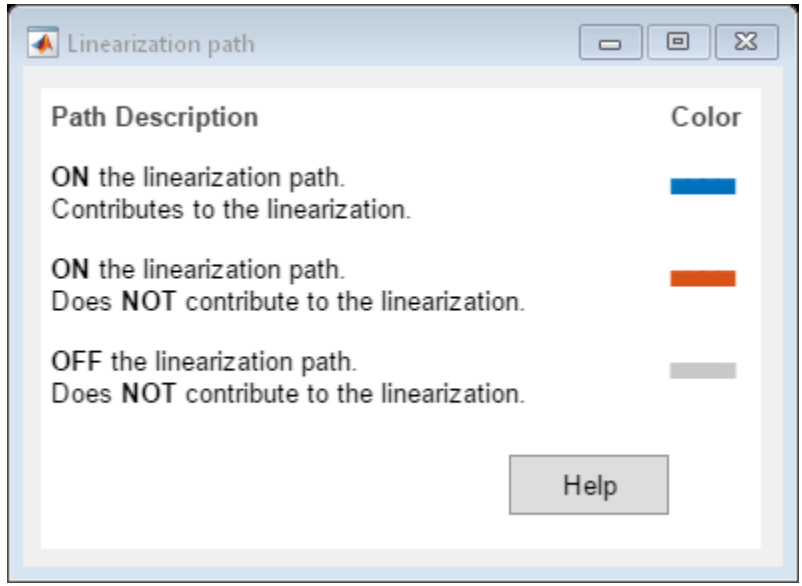
```
getBlockInfo(adv2)
```

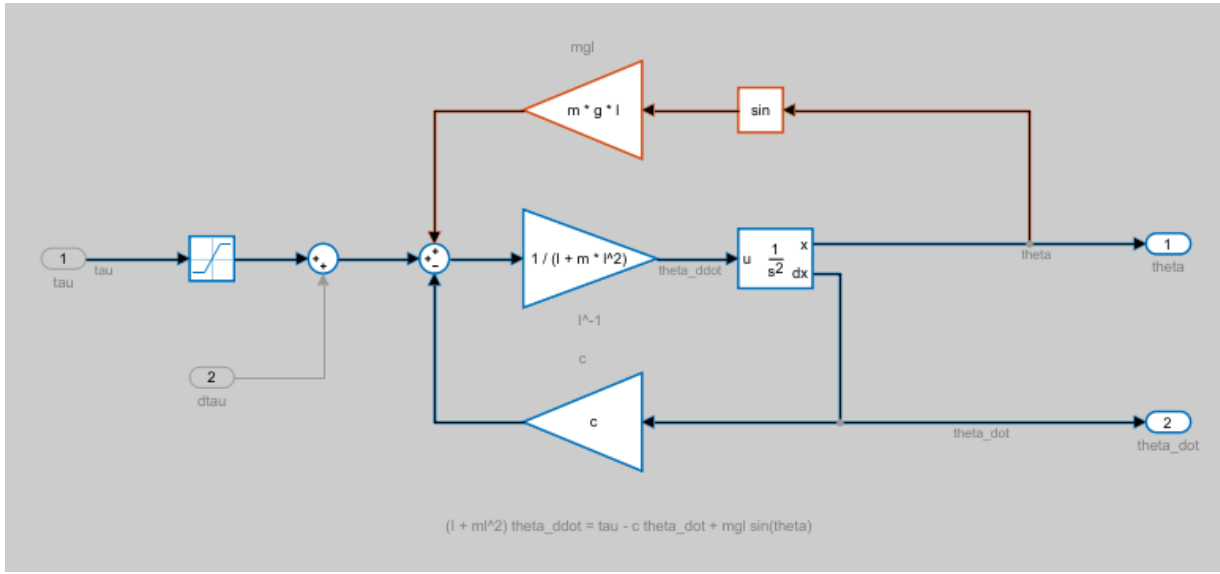
```
ans =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

The two `Trigonometric Function` blocks are still listed.

Highlight the linearization path for the updated linearization.

```
highlight(advisor2)
```





Most of the blocks are now contributing to the model linearization, except for the paths going through the listed `Trigonometric Function` blocks.

To understand why these blocks are not contributing to the linearization, view their corresponding block diagnostic information. For example, obtain the diagnostic information for the second `Trignonometric Function` block.

```
diag = getBlockInfo(adv2,2)
```

```
diag =
Linearization Diagnostics for scdpendulum/pendulum/Trigonometric Function with properti

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Perturbation'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

View the linearization of this block.

```
diag.Linearization
```

```
ans =

  D =
       u1
   y1   0

Name: Trigonometric Function
Static gain.
```

The block linearized to zero. To see if this result is expected for the current operating condition of the block, check its operating point.

```
diag.OperatingPoint
```

```
ans =
  BlockOperatingPoint
```

The input operating point of the block is $\pi/2 = 1.5708$.

You can find the linearization of the block analytically by taking the first derivative of the `sin` function with respect to the input:

$$\frac{\partial}{\partial u}\sin(u) = \cos(u)$$

Therefore, when evaluated at $u = \pi/2$ the linearization of the block is zero. The source of the input is the first output of the second-order integrator, which is dependent upon the state `theta`. Therefore, this block linearizes to zero if $\theta = \pi/2 + k\pi$, where k is an integer. The same condition applies for the other `Trigonometric Function` in the `angle_wrap` subsystem. If these blocks are not expected to linearize to zero, you can modify the operating point state `theta`, and relinearize the model.

Create and Run Custom Queries

The Linearization Advisor also provides objects and functions for creating custom queries. Using these queries, you can find blocks in your model that match specific criteria. For example, to find all SISO blocks that are linearized using numerical perturbation, first create query objects for each search criterion:

- Has one input
- Has one output
- Is numerically perturbed

```
qIn = linqueryHasInputs(1);
qOut = linqueryHasOutputs(1);
qPerturb = linqueryIsNumericallyPerturbed;
```

Create a `CompoundQuery` object by combining these query objects using logical operators.

```
sisopert = qIn & qOut & qPerturb;
```

Search the block diagnostics in `advisor2` for blocks matching these criteria.

```
sisopertBlocks = find(advisor2,sisopert)
```

```
sisopertBlocks =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: '((Has 1 Inputs & Has 1 Outputs) & Perturbation)'
```

There are three SISO blocks in the model that are linearized using numerical perturbation.

For more information on using custom queries, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Functions

`advise` | `find`

More About

- “Identify and Fix Common Linearization Issues” on page 4-8
- “Troubleshoot Linearization Results in Linear Analysis Tool” on page 4-21
- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52

Find Blocks in Linearization Results Matching Specific Criteria

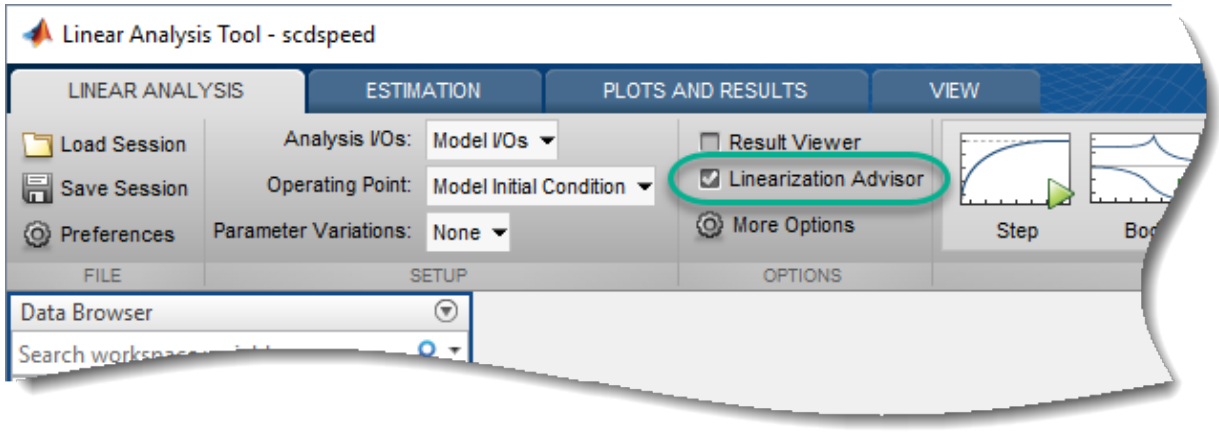
When you linearize a Simulink model, you can find blocks in your linearization result that match specific criteria using the Linearization Advisor. You can specify search criteria to find blocks that can:

- Potentially cause linearization issues in your model, if your model does not linearize as expected. For more information on identifying and fixing linearization issues using the Linearization Advisor, see “Identify and Fix Common Linearization Issues” on page 4-8.
- Help you gain insight into your model linearization, even if the model has linearized as expected.

You can also query the Linearization Advisor at the command line using the `find` function. For an example, see “Troubleshoot Linearization Results at Command Line” on page 4-40.

Searching the linearization results requires linearization diagnostic information. To collect this information, you must enable the Linearization Advisor before linearizing your model.

To enable the Linearization Advisor, in the Linear Analysis Tool, on the **Linear Analysis** tab, select **Linearization Advisor**.



When you select this option and linearize your model, the software opens an **Advisor** tab for troubleshooting your linearization results. You can then find blocks of interest in the linearization results by running queries with the Linearization Advisor.

After finding blocks of interest, you can examine the individual block linearizations using the linearization diagnostic information. For more information, see “Block Linearization Troubleshooting” on page 4-58.

Run Built-In Queries

The Linearization Advisor provides a set of built-in queries for searching your linearization results. These queries are useful for finding blocks that are potentially causing linearization issues. To run one of these queries, on the **Advisor** tab, in the **Queries** section, click the query.

Built-In Query	Find Blocks That...
Linearization Advice	Are potentially problematic for linearization. This query is performed by default when the **Advisor** tab opens.
Diagnostics on Path	Are on the linearization path and that have diagnostic messages regarding their linearization. This query is a subset of the `Linearization Advice` query.
Substitutions on Path	Are on the linearization path and have a custom block linearization specified. This query is a subset of the `Linearization Advice` query.
Zero I/O Pair on Path	Are on the linearization path and have at least one input/output pair that linearizes to zero.
All Blocks on Path	Are on the linearization path; that is, blocks where at least one linearization input is connected to at least one linearization output through the block.

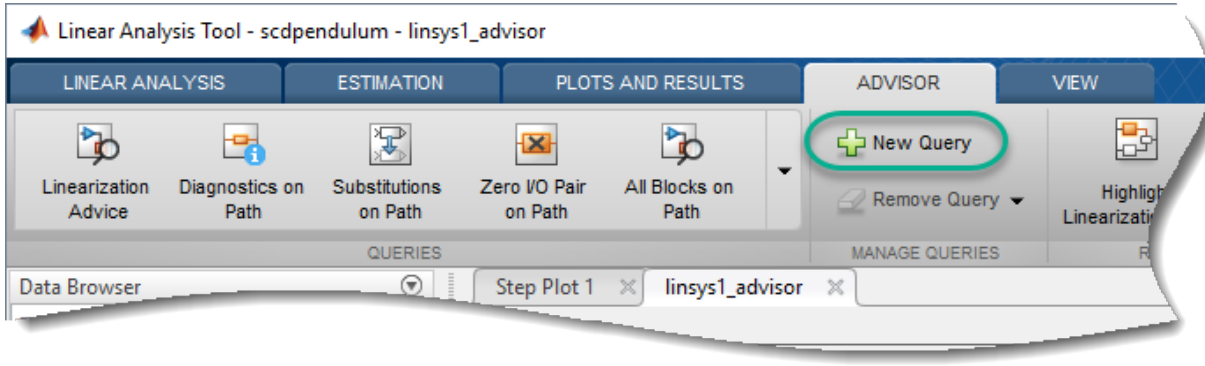
Create and Run Queries

The linearization advisor also provides a set of simple queries for searching your model. You can run these queries on their own or use them to create compound queries.

Simple Query	Find Blocks That...
`All Blocks`	Are in the linearized model.

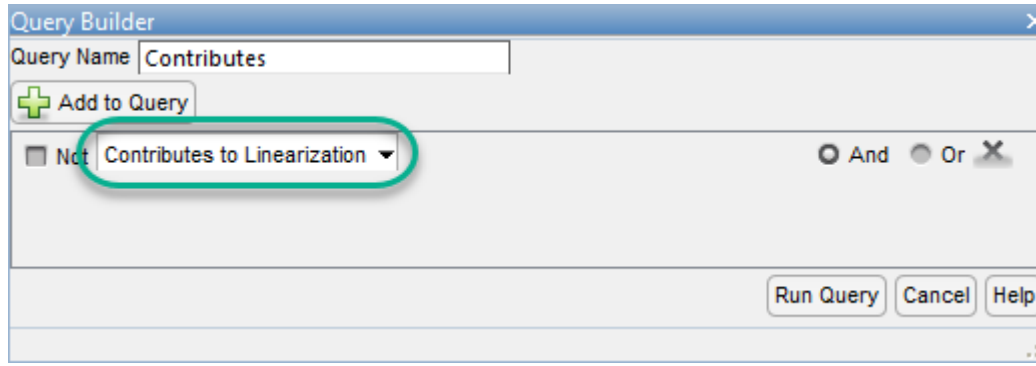
Simple Query	Find Blocks That...
`Linearized to Zero`	Linearize to zero.
`Block Substituted`	Have a custom block linearization specified.
`On Linearization Path`	Are on the linearization path.
`Contributes to Linearization`	Numerically contribute to the model linearization result.
`Exact`	Are linearized using their defined exact linearization.
`Perturbation`	Are linearized using numerical perturbation.
`Has Diagnostics`	Have diagnostic messages regarding their linearization.
`'BlockType' Blocks`	Are of a specified type.
`Has 'Nu' Inputs`	Have a specified number of inputs.
`Has 'Nx' States`	Have a specified number of states.
`Has 'Ny' Outputs`	Have a specified number of outputs.
`Has 'Ts' Sample Time`	Have a specified sample time.
`Has Zero I/O Pair`	Have at least one input/output pair that linearizes to zero.

To run a simple query, in the Linear Analysis Tool, on the **Advisor** tab, click **New Query**.



In the Query Builder dialog box, configure the query. For example, create a query for finding all blocks that numerically contribute to the linearization result.

1. In the **Query Name** field, specify the name for the query as `Contributes`.
2. In the drop-down list, select `Contributes to Linearization`.

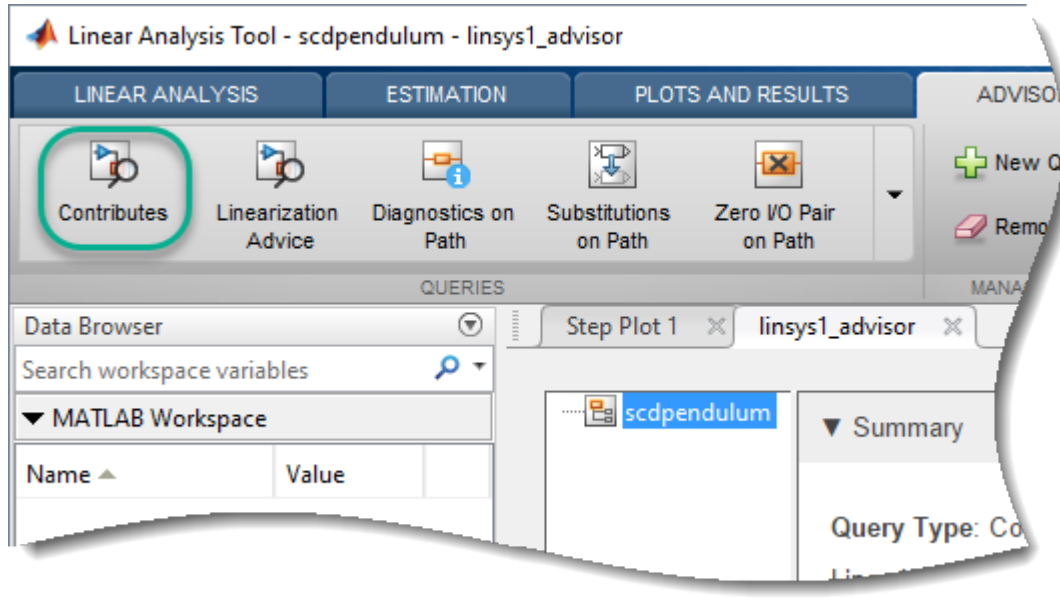


3. If you select any of the following queries, specify the corresponding search parameter.

Query	Search Parameter
`'BlockType' Blocks`	**Block Type** — This parameter corresponds to the `blocktype` property of the block. For more information, see `linqueryIsBlockType`.
`Has 'Nu' Inputs`	**Inputs** — Specify a positive integer.
`Has 'Nx' States`	**States** — Specify a positive integer.
`Has 'Ny' Outputs`	**Outputs** — Specify a positive integer.
`Has 'Ts' Sample Time`	**Sample Time** — Specify a nonzero scalar. To find continuous-time blocks, specify `0`.

4. To create and run the query, click **Run Query**. The software runs the query and, on the **Advisor** tab, displays the list of blocks that contribute to the model linearization.

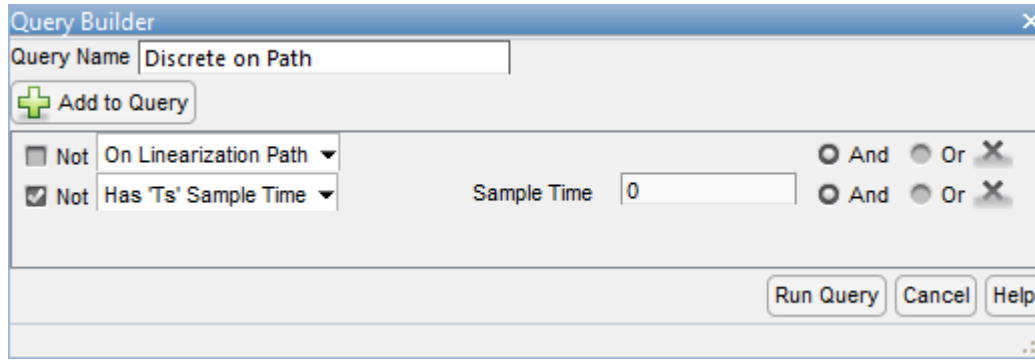
   The query is added to the **Queries** section.



You can also create compound queries by logically combining existing queries using **And**, **Or**, and **Not** logical operations. You can create a compound query using simple queries, built-in queries, or other compound queries.

To create a compound query, in the Query Builder dialog box, configure the query using multiple search criteria. For example, create a query to find all discrete-time blocks that are on the linearization path.

1. In the **Query Name** field, specify the name for the query as `Discrete on Path`.
2. To find blocks on the linearization path, in the drop-down list, select `On Linearization Path`.
3. To add another search criteria, click **Add to Query**. The software adds a second row to the search criteria. By default, the search criteria are combined using an **And** operation.
4. To find discrete-time blocks, first add a search criteria to find continuous-time blocks. In the second row, in the drop-down list, select `Has 'Ts' Sample Time`. Keep the default **Sample Time** of `0`.
5. To find discrete-time blocks, in the second row, select **Not**.



6 Click **Run Query**.

Each time you create a custom query, the software adds it to the drop-down list of search criteria in the Query Builder dialog box. You can then use your custom queries to create more complex queries. For example, to find discrete-time blocks on the linearization path that are linearized using numerical perturbation, create a query that combines the `Discrete on Path` custom query with the `Perturbed` simple query using an **And** operation.

See Also

Apps

Linear Analysis Tool

Functions

`find`

More About

- “Identify and Fix Common Linearization Issues” on page 4-8
- “Troubleshoot Linearization Results in Linear Analysis Tool” on page 4-21
- “Troubleshoot Linearization Results at Command Line” on page 4-40

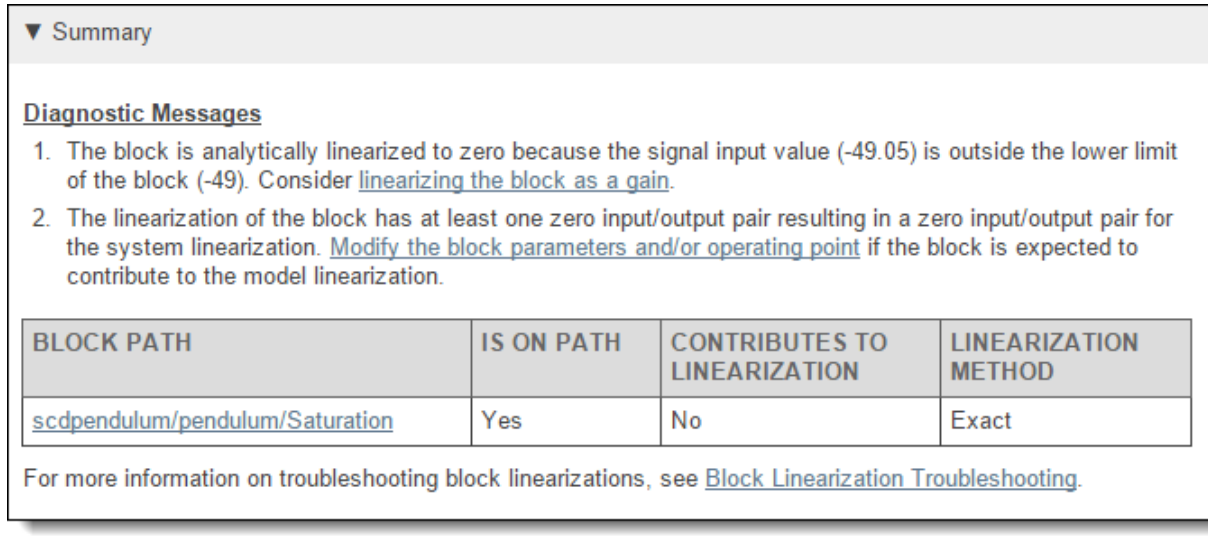
Block Linearization Troubleshooting

Once you identify blocks of interest in the linearization results for your Simulink model by querying the Linearization Advisor, you can troubleshoot the individual block linearizations. For more information on querying the Linearization Advisor and viewing block diagnostic information, see “Identify and Fix Common Linearization Issues” on page 4-8.

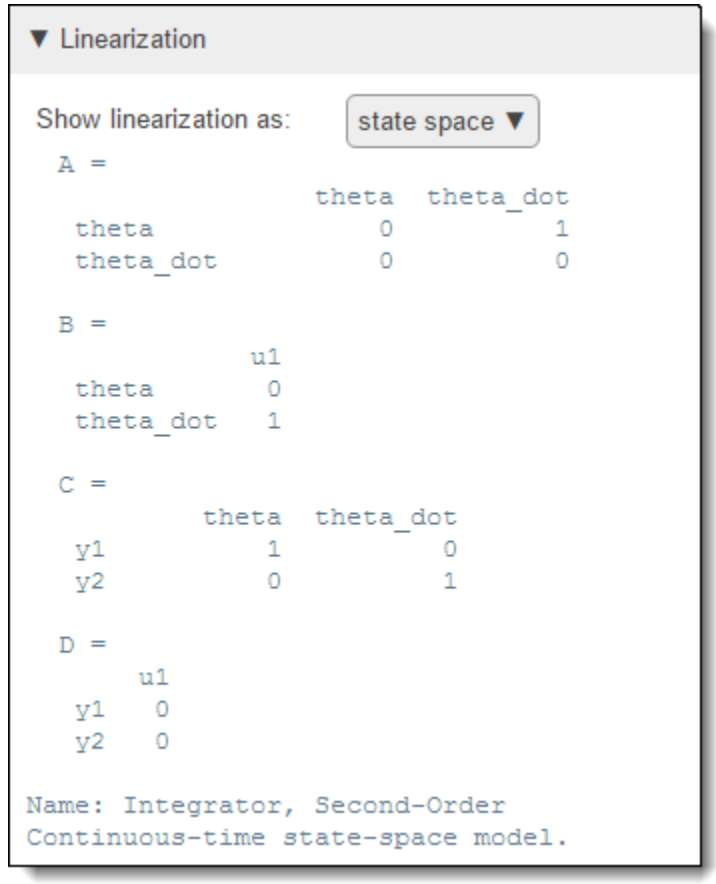
You can also troubleshoot individual block linearizations at the command line using a `BlockDiagnostic` object. For an example, see “Troubleshoot Linearization Results at Command Line” on page 4-40.

In the Linear Analysis Tool, on the **Advisor** tab, the detailed diagnostic information for a block linearization shows:

- A diagnostic summary, showing any corresponding diagnostic messages and a linearization summary table.



- The block linearization value.



- The block operating point; the state and input values for which the block is linearized.

▼ Operating Point

STATES	x
theta	1.5708
theta_dot	0
PORT	u
1	0.0090909

You can diagnose potential linearization issues using this information.

Diagnostic Messages

Linearization diagnostic messages indicate blocks with properties or linearizations that correspond to common linearization problems. Fixing linearization issues identified in diagnostic messages is a good first step when troubleshooting your linearization.

Some block configurations that can generate diagnostic messages include:

- Blocks with nondouble input or output signals and no predefined exact linearization. Such blocks linearize to zero and generate diagnostic messages.
- Discontinuous blocks linearized at an operating point near a discontinuity. If such blocks are not treated as a gain during linearization, the software generates diagnostic messages regarding their linearization.
- Blocks with least one input/output pair that linearizes to zero and that causes a zero input/output pair in the overall model linearization. A linearization has a zero input/output pair when a change in an input signal value does not produce a corresponding change in an output value.
- Blocks that do not support linearization because they do not have a predefined exact linearization and do not support numerical perturbation.

Some diagnostic messages propose solutions to their corresponding linearization issues. For example, when an input signal is outside the saturation limits of a Saturation block, the diagnostic message proposes treating the block as a gain during linearization.

Linearization Summary

The linearization summary table displays the following properties of the block linearization:

- **Block Path** — Location of the block in the Simulink model. To highlight the block in the model, click the block path.
- **Is On Path** — Flag indicating whether the block is on the linearization path; that is, at least one linearization input is connected to at least one linearization output through the block. If you expect a block to be on the linearization path and it is not on the path, check the analysis point configuration in your model. Incorrectly placed linearization I/Os or loop openings can exclude blocks from the linearization path. Similarly, placing incorrect analysis points can unexpectedly add blocks to the linearization path.
- **Contributes to Linearization** — Flag indicating whether the block numerically contributes to the overall model linearization. If a block unexpectedly does not contribute to the linearization result, investigate the linearization of the block and other blocks in the same branch of the linearization path. For example, if an adjacent block on the linearization path linearizes to zero, an otherwise correctly linearized block can be excluded from the linearization result.
- **Linearization method** — The method used to linearize the model, specified as one of the following:
  - **Exact** — The block linearization is computed using the defined analytic Jacobian of the block.
  - **Perturbation** — The block does not have an analytic Jacobian. Instead, the block is linearized using numerical perturbation of its inputs and states. Some numerically perturbed blocks, such as those with discontinuities or nondouble input signals can linearize to zero.
  - **Block Substituted** — The block linearization is specified using a custom block linearization. Consider checking that the specified block linearization is correct for your application. For more information, see “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-158 and “Specify D-Matrix System for Block Linearization Using Function” on page 2-159.
  - **Simscape Engine** — The block diagnostics correspond to a Simscape network in your model. For more information on linearizing and troubleshooting Simscape networks, see “Linearize Simscape Networks” on page 2-194.
  - **Not Supported** — The block does not have an analytic Jacobian and does not support numerical perturbation. Specify the linearization for this block using a

custom linearization. For more information, see “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-158 and “Specify D-Matrix System for Block Linearization Using Function” on page 2-159.

Block Linearization

To verify whether a block linearized as expected, check the block linearization equations. By default the software displays the linearization in state-space format. In the **Show linearization as** drop-down list, you can select a different display format.

To diagnose the cause of an unexpected block linearization, such as a block that linearizes to zero, consider:

- Any corresponding diagnostic messages. These messages can highlight common causes of incorrect linearizations and propose potential solutions.
- The block operating point. For example, if the input to a saturation block is outside the saturation limits of the block, the block linearizes to zero.
- The block parameters. For example, if a block is configured to use nondouble inputs or states and has no predefined exact linearization, it linearizes to zero.

Block Operating Point

If the block does not linearize as expected, check the operating point. The operating point at which the block is linearized consists of input and state values. If the operating point for the block is incorrect, check whether the overall model operating point is correct. For more information, see “Check Operating Point” on page 4-5.

If an input signal value in the block operating point is incorrect, investigate the linearization of upstream blocks from that signal. For example, consider a Product block with two inputs. The operating point of this block consists of the two input signal values. If either input value is zero, the path from the other input to the output linearizes to zero.

If you expect the Product block to contribute to the linearization result for the operating point at which you linearized the model, check the linearization for the block that generates the zero input signal. For complex models, the cause of the incorrect input signal can be more than one block upstream.

Common Problematic Blocks

Some Simulink blocks have properties that cause them to linearize poorly. Often, such blocks either linearize to zero or have linearization diagnostic messages associated with them. Therefore, the Linearization Advisor identifies them as potentially problematic blocks when the **Advisor** tab first opens.

The following table shows some blocks that commonly cause linearization issues and proposes potential fixes for each block. All these blocks have corresponding diagnostic messages.

Block Type	Linearization Issue	Possible Fix
Blocks that do not support linearization	Some blocks are implemented without defined analytic Jacobians and do not support numerical perturbation.	Specify a custom block linearization. For examples, see "Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-158 and "Specify D-Matrix System for Block Linearization Using Function" on page 2-159.

Block Type	Linearization Issue	Possible Fix
Blocks with discontinuities	Blocks with discontinuities typically have poor linearization results when the operating point is near the discontinuity.	• Treat the block as a gain of `1` during linearization. To do so, select the **Treat as gain when linearizing** block parameter. • Specify a custom block linearization. For examples, see "Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-158 and "Specify D-Matrix System for Block Linearization Using Function" on page 2-159.
Event-Based Subsystems (triggered subsystems)	Blocks within event-based subsystems linearize to zero because such subsystems do not trigger during linearization.	When possible, specify a custom event-based subsystem linearization as a lumped average model or periodic function call subsystem. For more information, see "Linearize Event-Based Subsystems (Externally Scheduled Subsystems)" on page 2-185.

Block Type	Linearization Issue	Possible Fix
Blocks with nondouble-precision signals	Blocks that have nondouble-precision inputs signals or states and do not have defined analytic Jacobians linearize to zero.	Convert the nondouble-precision data types to double-precision. For more information, see "Linearize Blocks with Nondouble Precision Data Type Signals" on page 2-183.
Blocks that linearize using numerical perturbation rather than defined analytic Jacobians	Blocks that are located near discontinuous regions, such as S-Functions, MATLAB function blocks, or lookup tables, are sensitive to numerical perturbation levels. If the perturbation level is too small, the block linearizes to zero.	Change the numerical perturbation level of the block. For more information, see "Change Perturbation Level of Blocks Perturbed During Linearization" on page 2-181.

See Also

More About

- "Linearization Troubleshooting Overview" on page 4-2
- "Identify and Fix Common Linearization Issues" on page 4-8

Speed Up Linearization of Complex Models

In this section...

Factors That Impact Linearization Performance

Large Simulink models and blocks with complex initialization functions can slow linearization.

Usually, the time it takes to linearize a model is directly related to the time it takes to update the block diagram (Simulink).

Blocks with Complex Initialization Functions

Use the MATLAB Profiler to identify complex bottlenecks in block initialization functions.

In the MATLAB Profiler, run the command:

```
set_param(modelname,'SimulationCommand','update')
```

Disabling the Linearization Advisor in the Linear Analysis Tool

You can speed up the linearization of large models by disabling the Linearization Advisor in the Linear Analysis Tool.

The Linearization Advisor stores diagnostic information, including linearization values of individual blocks, which can impact linearization performance.

To disable the Linearization Advisor, in the Linear Analysis Tool, on the **Linear Analysis** tab, clear **Linearization Advisor**.

Tip Alternatively, you can disable the Linearization Advisor globally in the Simulink Control Design tab of the MATLAB preferences dialog box. Clear the **Launch**

Linearization for exact linearizations in the linear analysis tool check box. This global preference persists from session to session until you change this preference.

Batch Linearization of Large Simulink Models

When batch linearizing a large model that contains only a few varying parameters, you can use `linlftfold` to reduce the computational load.

See “More Efficient Batch Linearization Varying Parameters”.

See Also

Frequency Response Estimation

What Is a Frequency Response Model?

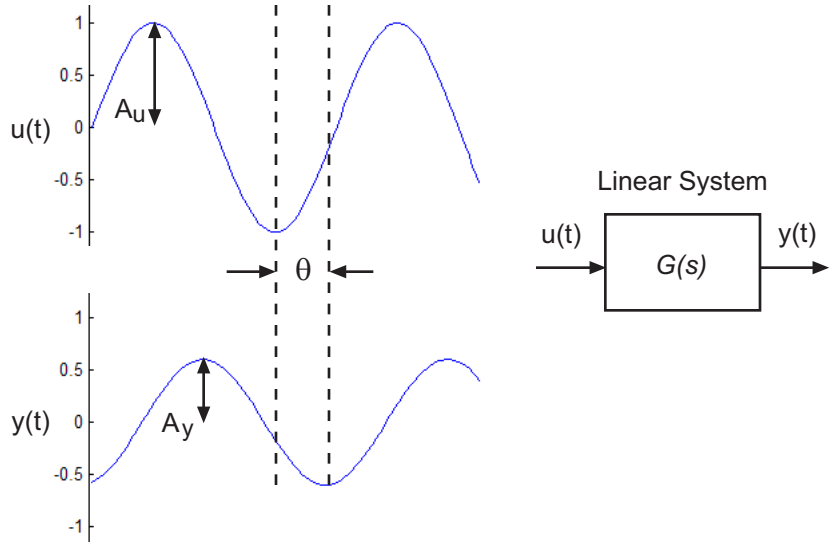
Frequency response describes the steady-state response of a system to sinusoidal inputs.

For a linear system, a sinusoidal input of frequency ω:

$$u(t) = A_u \sin \omega t$$

results in an output that is also a sinusoid with the same frequency, but with a different amplitude and phase θ:

$$y(t) = A_y \sin(\omega t + \theta)$$



Frequency response $G(s)$ for a stable system describes the amplitude change and phase shift as a function of frequency:

$$G(s) = \frac{Y(s)}{U(s)}$$

$$|G(s)| = |G(j\omega)| = \frac{A_y}{A_u}$$

$$\theta = \angle \frac{Y(j\omega)}{X(j\omega)} = \tan^{-1}\left(\frac{\text{imaginary part of } G(j\omega)}{\text{real part of } G(j\omega)} \right)$$

where $Y(s)$ and $U(s)$ are the Laplace transforms of $y(t)$ and $u(t)$, respectively.

Frequency Response Model Applications

You can estimate the frequency response of a Simulink model as a frequency response model (`frd` object), without modifying your Simulink model.

Applications of frequency response models include:

- Validating exact linearization results.

  Frequency response estimation uses a different algorithm to compute a linear model approximation and serves as an independent test of exact linearization. See "Validate Linearization In Frequency Domain" on page 2-140.
- Analyzing linear model dynamics.

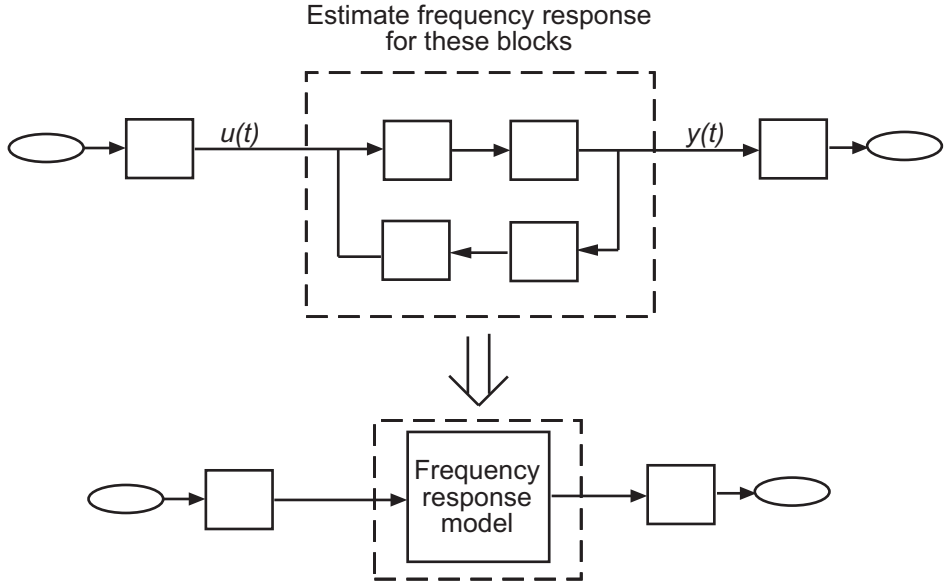
  Designing controller for the plant represented by the estimated frequency response using Control System Toolbox software.
- Estimating parametric models.

  See "Estimate Frequency Response Models with Noise Using System Identification Toolbox" on page 5-75.

Model Requirements

You can estimate the frequency response of one or more blocks in a stable Simulink model at steady state.

Your model can contain any Simulink blocks, including blocks with event-based dynamics. Examples of blocks with event-based dynamics include Stateflow charts, triggered subsystems, pulse width modulation (PWM) signals.



You should disable the following types of blocks before estimation:

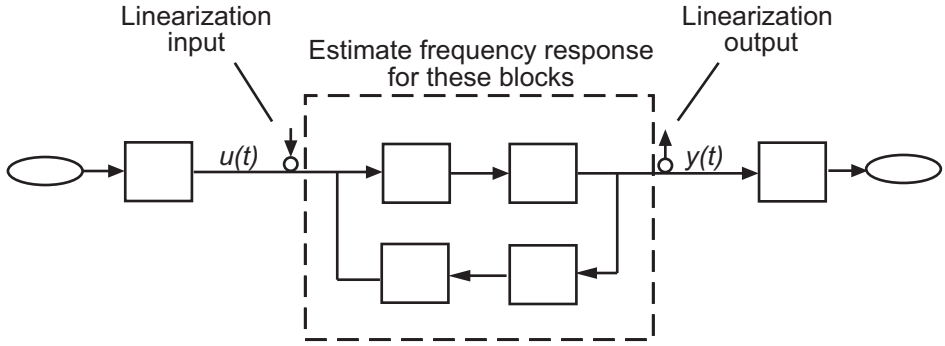
- Blocks that simulate random disturbances (noise).

  For alternatives ways to model systems with noise, see “Estimate Frequency Response Models with Noise Using Signal Processing Toolbox” on page 5-73.
- Source blocks that generate time-varying outputs that interfere with the estimation. See “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-58.

Estimation Requires Input and Output Signals

Frequency response estimation requires an input signal at the linearization input point to excite the model at frequencies of interest, such as a chirp or sinestream signal. A *sinestream* input signal is a series of sinusoids, where each sine wave excites the system for a period of time. You can inject the input signal anywhere in your model and log the simulated output, without having to modify your model.

Frequency response estimation adds the input signal you design to the existing Simulink signals at the linearization input point, and simulates the model to obtain the output at the linearization output point. For more information about supported input signals and their impact on the estimation algorithm, see "Estimation Input Signals" on page 5-7.



For multiple-input multiple-output (MIMO) systems, frequency response estimation injects the signal at each input channel separately to simulate the corresponding output signals. The estimation algorithm uses the inputs and the simulated outputs to compute the MIMO frequency response. If you want to inject different input signal at the linearization input points of a multiple-input system, treat your system as separate single-input systems. Perform independent frequency response estimations for each linearization input point using `frestimate`, and concatenate your frequency response results.

Frequency response estimation correctly handles open-loop linearization input and output points. For example, if the input linearization point is open, the input signal you design adds to the constant operating point value. The operating point is the initial output of the block with a loop opening.

The estimated frequency response is related to the input and output signals as:

$$G(s) \approx \frac{\text{fast Fourier transform of } y_{est}(t)}{\text{fast Fourier transform } u_{est}(t)}$$

where $u_{est}(t)$ is the injected input signal and $y_{est}(t)$ is the corresponding simulated output signal.

See Also

More About

- “Estimation Input Signals” on page 5-7

Estimation Input Signals

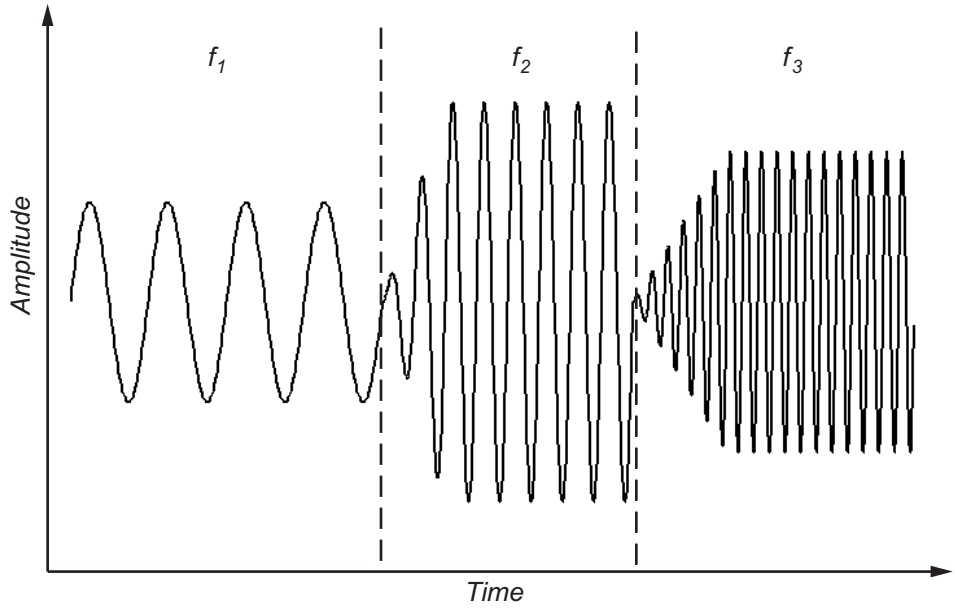
Frequency response estimation uses either sinestream or chirp input signals.

Sinusoidal Signal	When to Use
Sinestream	Recommended for most situations. Especially useful when: • Your system contains strong nonlinearities. • You require highly accurate frequency response models.
Chirp	Useful when: • Your system is nearly linear in the simulation range. • You want to quickly obtain a response for a lot of frequency points.

In this section...
“What Is a Sinestream Signal?” on page 5-7 “What Is a Chirp Signal?” on page 5-12

What Is a Sinestream Signal?

A *sinestream* signal consists of several adjacent sine waves of varying frequencies. Each frequency excites the system for a period of time.

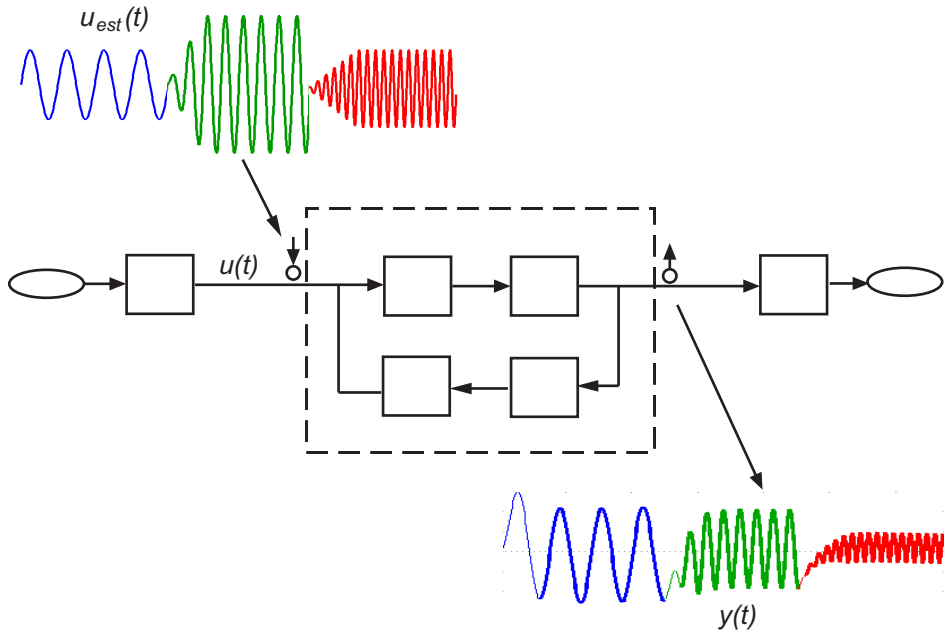


How Frequency Response Estimation Treats Sinestream Inputs

Frequency response estimation using `frestimate` performs the following operations on a sinestream input signal:

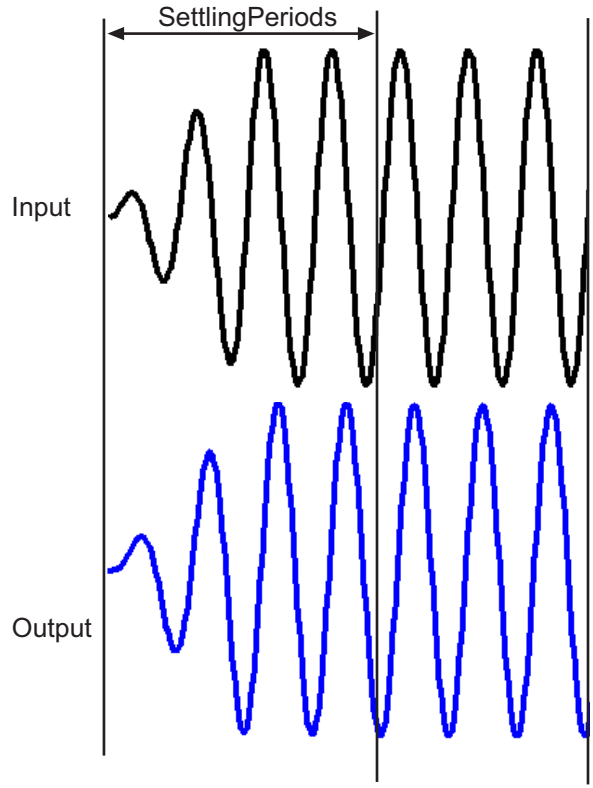
1. Injects the sinestream input signal you design, $u_{est}(t)$, at the linearization input point.
2. Simulates the output at the linearization output point.

   `frestimate` adds the signal you design to existing Simulink signals at the linearization input point.



3 Discards the `SettlingPeriods` portion of the output (and the corresponding input) at each frequency.

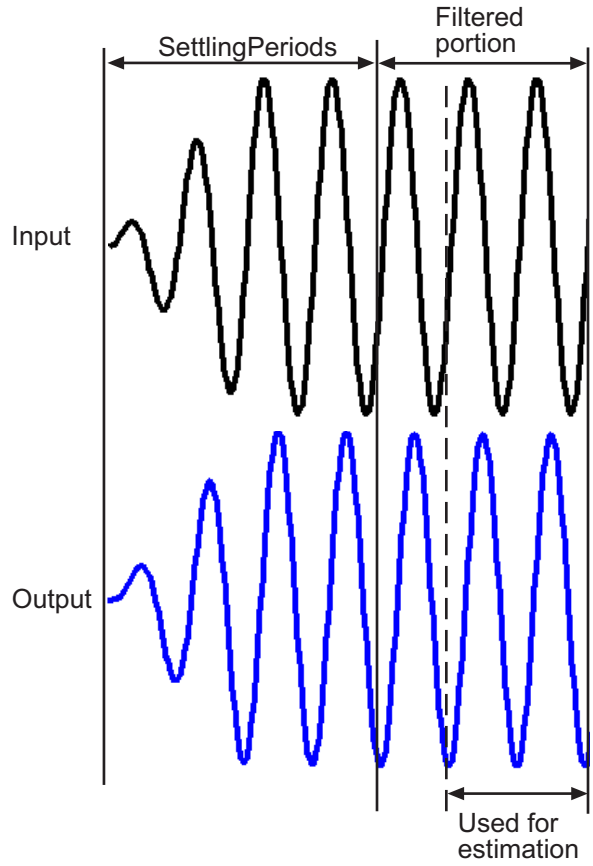
The simulated output at each frequency has a transient portion and steady state portion. `SettlingPeriods` corresponds to the transient components of the output and input signals. The periods following `SettlingPeriods` are considered to be at steady state.



4 Filters the remaining portion of the output and the corresponding input signals at each input frequency using a bandpass filter.

When a model is not at steady state, the response contains low-frequency transient behavior. Filtering typically improves the accuracy of your model by removing the effects of frequencies other than the input frequencies. These frequencies are problematic when your sampled data has finite length. These effects are called *spectral leakage*.

`frestimate` uses a finite impulse response (FIR) filter. The software sets the filter order to match the number of samples in a period such that any transients associated with filtering appear only in the first period of the filtered steady-state output. After filtering, `frestimate` discards the first period of the input and output signals.



You can specify to disable filtering during estimation using the signal `ApplyFilteringInFRESTIMATE` property.

5 Estimates the frequency response of the processed signal by computing the ratio of the fast Fourier transform of the filtered steady-state portion of the output signal $y_{est}(t)$ and the fast Fourier transform of the filtered input signal $u_{est}(t)$:

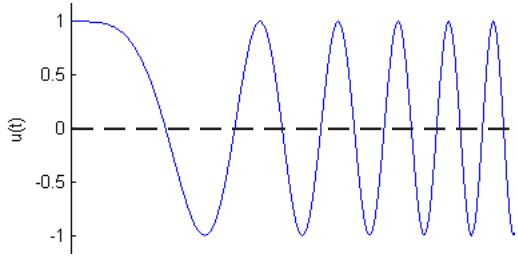
$$G(s) \approx \frac{\text{fast Fourier transform of } y_{est}(t)}{\text{fast Fourier transform } u_{est}(t)}$$

To compute the response at each frequency, `frestimate` uses only the simulation output at that frequency.

What Is a Chirp Signal?

The swept-frequency cosine (chirp) input signal excites your system at a range of frequencies, such that the input frequency changes instantaneously.

Alternatively, you can use the sinestream signal on page 5-13, which excites the system at each frequency for several periods.



See Also

More About

- “Create Sinestream Input Signals” on page 5-13
- “Create Chirp Input Signals” on page 5-19

Create Sinestream Input Signals

In this section...

Create Sinestream Signals Using Linear Analysis Tool

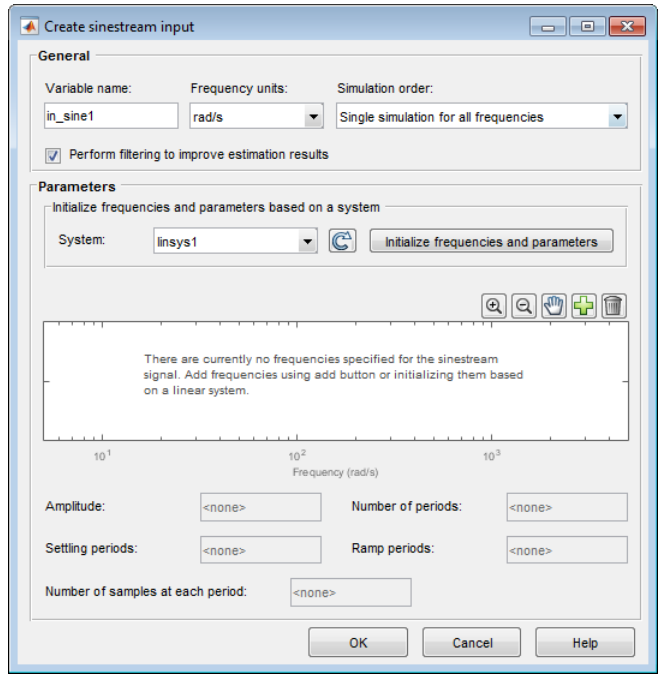
This example shows how to create a sinestream input signal based upon a linearized model using the Linear Analysis Tool. If you do not have a linearized model in your workspace, you can manually construct a sinestream as shown in “Estimate Frequency Response Using Linear Analysis Tool” on page 5-26.

1 Obtain a linearized model, `linsys1`.

  For example, see “Linearize Simulink Model at Model Operating Point” on page 2-69, which shows how to linearize a model.

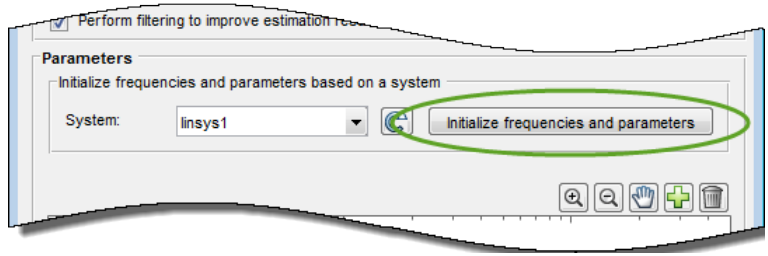
2 In the Linear Analysis Tool, in the **Estimation** tab, in the **Input Signal** drop-down list, select **Sinestream**.

  The Create sinestream input dialog box opens.

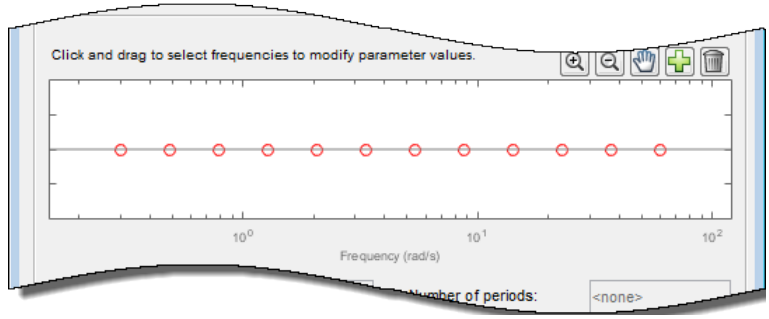


Note Selecting **Sinestream** creates a continuous-time signal. To generate a discrete-time signal, in the **Input Signal** drop-down list, select **Fixed Sample Time Sinestream**.

3 In the **System** list, select `linsys1`. Click **Initialize frequencies and parameters**.

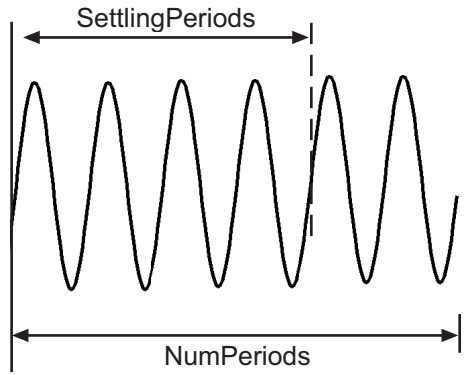


This action adds frequency points to the Frequency content viewer.

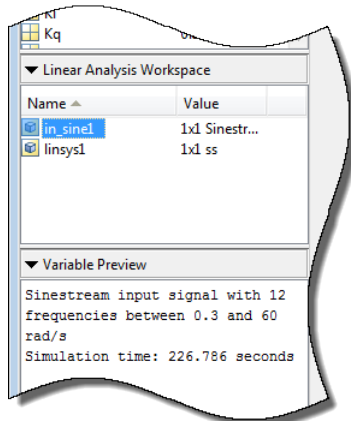


The software automatically selects frequency points based on the dynamics of `linsys1`. The software also automatically determines other parameters of the sinestream signal, including:

- Amplitude
- Number of periods
- Settling periods
- Ramp periods
- Number of samples at each period



4 Click **OK** to create the sinestream input signal. A new input signal, `in_sine1`, appears in the **Linear Analysis Workspace**.



Create Sinestream Signals Using MATLAB Code

You can create a sinestream signal from both continuous-time and discrete-time signals in Simulink models using the following commands:

Signal at Input Linearization Point	Command
Continuous	`frest.Sinestream`
Discrete	`frest.createFixedTsSinestream`

Create a sinestream signal in the most efficient way using a linear model that accurately represents your system dynamics:

```
input = frest.Sinestream(sys)
```

`sys` is the linear model you obtained using exact linearization.

You can also define a linear system based on your insight about the system using the `tf`, `zpk`, and `ss` commands.

For example, create a sinestream signal from a linearized model:

```
magball
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',...
              1,'output');
```

```
sys = linearize('magball',io);
input = frest.Sinestream(sys)
```

The resulting `input` signal stores the frequency values as `Frequency`. `frest.Sinestream` automatically specifies `NumPeriods` and `SettlingPeriods` for each frequency:

```
Frequency           : [0.05786;0.092031;0.14638 ...] (rad/s)
Amplitude           : 1e-005
SamplesPerPeriod    : 40
NumPeriods          : [4;4;4;4 ...]
RampPeriods         : 0
FreqUnits (rad/s,Hz): rad/s
SettlingPeriods     : [1;1;1;1 ...]
ApplyFilteringInFRESTIMATE (on/off)    : on
SimulationOrder (Sequential/OneAtATime): Sequential
```

You can plot your input signal using `plot(input)`.

For more information about sinestream options, see the `frest.Sinestream` reference page.

The mapping between the parameters of the Create sinestream input dialog box in the Linear Analysis Tool and the options of `frest.Sinestream` is as follows:

Create sinestream input dialog box	`frest.Sinestream` option
Amplitude	`'Amplitude'`
Number of periods	`'NumPeriods'`
Settling periods	`'SettlingPeriods'`
Ramp periods	`'RampPeriods'`
Number of samples at each period	`'SamplesPerPeriod'`

See Also

`frest.Sinestream` | `frest.createFixedTsSinestream`

More About

- “Estimation Input Signals” on page 5-7

- “Create Chirp Input Signals” on page 5-19
- “Modify Estimation Input Signals” on page 5-23

Create Chirp Input Signals

In this section...

Create Chirp Signals Using Linear Analysis Tool

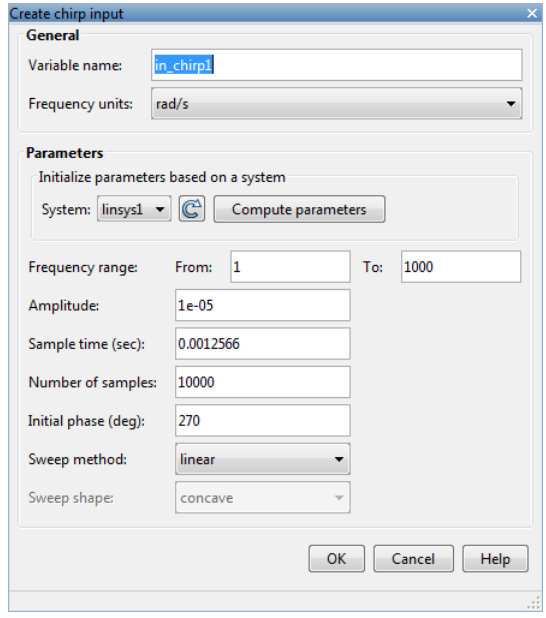
This example shows how to create a chirp input signal based upon a linearized model using the Linear Analysis Tool.

1 Obtain a linearized model, `linsys1`.

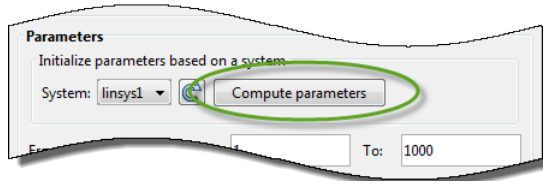
For example, see “Linearize Simulink Model at Model Operating Point” on page 2-69, which shows how to linearize a model.

2 In the Linear Analysis Tool, in the **Estimation** tab, in the **Input Signal** drop-down list, select **Chirp**.

The Create chirp input dialog box opens.



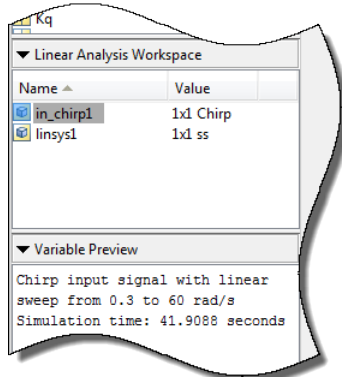
3 In the **System** list, select `linsys1`. Click **Compute parameters**.



The software automatically selects frequency points based on the dynamics of `linsys1`. The software also automatically determines other parameters of the chirp signal, including:

- frequency range at which the linear system has interesting dynamics (see the **From** and **To** boxes of **Frequency Range**).
- amplitude.
- sample time. To avoid aliasing, the Nyquist frequency of the signal is five times the upper end of the frequency range, $\frac{2\pi}{5 * \max(FreqRange)}$.
- number of samples.
- initial phase.
- sweep method
- sweep shape.

4 Click **OK** to create the chirp input signal. A new input signal `in_chirp1` appears in the **Linear Analysis Workspace**.



Create Chirp Signals Using MATLAB Code

Create a chirp signal in the most efficient way using a linear model that accurately represents your system dynamics:

```
input = frest.Chirp(sys)
```

`sys` can be the linear model you obtained using exact linearization techniques. You can also define a linear system based on your insight about the system using the `tf`, `zpk`, and `ss` commands.

For example, create a chirp signal from a linearized model:

```
magball
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',...
               1,'output');
sys = linearize('magball',io);
input = frest.Chirp(sys)
```

The `input` signal is:

```
      FreqRange              : [0.0578598408615998 10065.3895573969] (rad/s)
      Amplitude              : 1e-005
      Ts                     : 0.00012484733494616 (sec)
      NumSamples             : 1739616
      InitialPhase           : 270 (deg)
      FreqUnits (rad/s or Hz): rad/s
      SweepMethod(linear/    : linear
                  quadratic/
                  logarithmic)
```

You can plot your input signal using `plot(input)`.

For more information about chirp signal properties, see the `frest.Chirp` reference page.

The mapping between the parameters of the Create chirp input dialog box in the Linear Analysis Tool and the options of `frest.Chirp` is as follows:

Create chirp input dialog box	`frest.Chirp` option
Frequency range > From	First element associated with the `'FreqRange'` option

Create chirp input dialog box	`frest.Chirp` option
Frequency range > To	Second element associated with the `'FreqRange'` option
Amplitude	`'Amplitude'`
Sample time (sec)	`'Ts'`
Number of samples	`'NumSamples'`
Initial phase (deg)	`'InitialPhase'`
Sweep method	`'SweepMethod'`
Sweep shape	`'Shape'`

See Also

More About

- “Estimation Input Signals” on page 5-7
- “Create Sinestream Input Signals” on page 5-13

Modify Estimation Input Signals

When the frequency response estimation produces unexpected results, you can try modifying the input signal properties in the ways described in “Troubleshooting Frequency Response Estimation” on page 5-46.

Modify Sinestream Signal Using Linear Analysis Tool

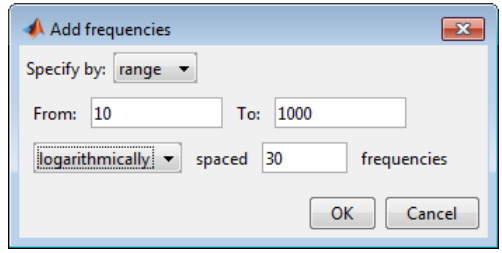
Add Frequency Points to Sinestream Input Signal

This example shows how to add frequency points to an existing sinestream input signal using the Linear Analysis Tool.

1. Create a sinestream input signal, `in_sine1`, as shown in “Create Sinestream Signals Using Linear Analysis Tool” on page 5-13.
2. Double-click `in_sine1` in the Linear Analysis Workspace area of the Linear Analysis Tool.

   The Edit sinestream dialog box opens.
3. In the Frequency content viewer, click  in the Frequency content toolbar.

   The Add frequencies dialog box opens.



4. Enter the frequency range of the points to be added.
5. Click **OK** to add the specified frequency points to `in_sine1`.

Delete Frequency Point from Sinestream Input Signal

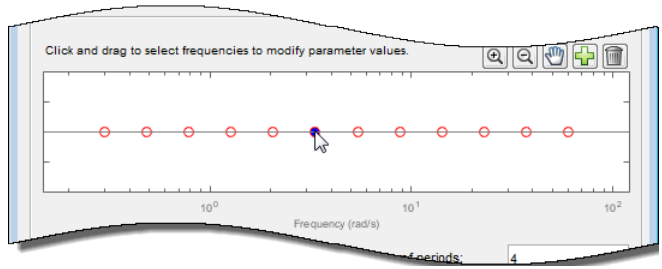
This example shows how to delete frequency points from an existing sinestream input signal using the Linear Analysis Tool.

1 Create a sinestream input signal, `in_sine1`, as shown in "Create Sinestream Signals Using Linear Analysis Tool" on page 5-13.

2 Double-click `in_sine1` in the Linear Analysis Workspace area of the Linear Analysis Tool.

The Edit sinestream dialog box opens.

3 In the Frequency content viewer, select the frequency point to delete.



The selected point appears blue.

Tip To select multiple frequency points, click and drag across the frequency points of interest.

4 Click  in the Frequency content toolbar to delete the selected frequency point(s) from the Frequency content viewer.

5 Click **OK** to save the modified input signal.

Modify Parameters for a Frequency Point in Sinestream Input Signal

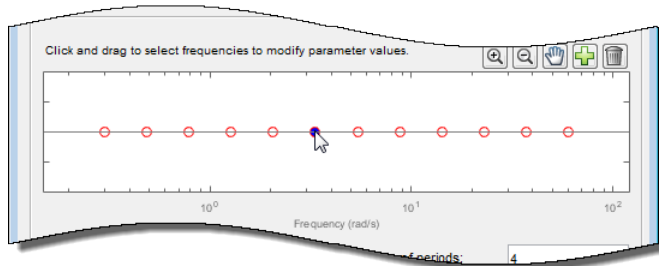
This example shows how to modify signal parameters of an existing sinestream input signal using the Linear Analysis Tool.

1 Create a sinestream input signal, `in_sine1`, as shown in "Create Sinestream Signals Using Linear Analysis Tool" on page 5-13.

2 Double-click `in_sine1` in the Linear Analysis Workspace area of the Linear Analysis Tool.

The Edit sinestream dialog box opens.

3 In the Frequency content viewer, select the frequency point(s) to delete.



The selected point(s) appears blue.

Tip To select multiple frequency points, click and drag across the frequency points of interest.

4 Enter the new values for the signal parameters.

If the parameter value is `<mixedvalue>`, the parameter has different values for some of the frequency points selected.

5 Click **OK** to save the modified input signal.

Modify Sinestream Signal Using MATLAB Code

For example, suppose that you used a sinestream input signal, and the output at a specific frequency did not reach steady state. In this case, you can modify the characteristics of the sinestream input at the corresponding frequency.

```
input.NumPeriods(index) = NewNumPeriods;
input.SettlingPeriods(index) = NewSettlingPeriods;
```

where `index` is the frequency value index of the sine wave you want to modify. `NewNumPeriods` and `NewSettlingPeriods` are the new values of `NumPeriods` and `SettlingPeriods`, respectively.

To modify several signal properties at a time, you can use the `set` command. For example:

```
input = set(input,'NumPeriods',NewNumPeriods,...
                  'SettlingPeriods',NewSettlingPeriods)
```

After modifying the input signal, repeat the estimation.

Estimate Frequency Response Using Linear Analysis Tool

This example shows how to estimate the frequency response of a Simulink model using the Linear Analysis Tool.

Open Simulink model and Linear Analysis Tool.

1 Open Simulink model.

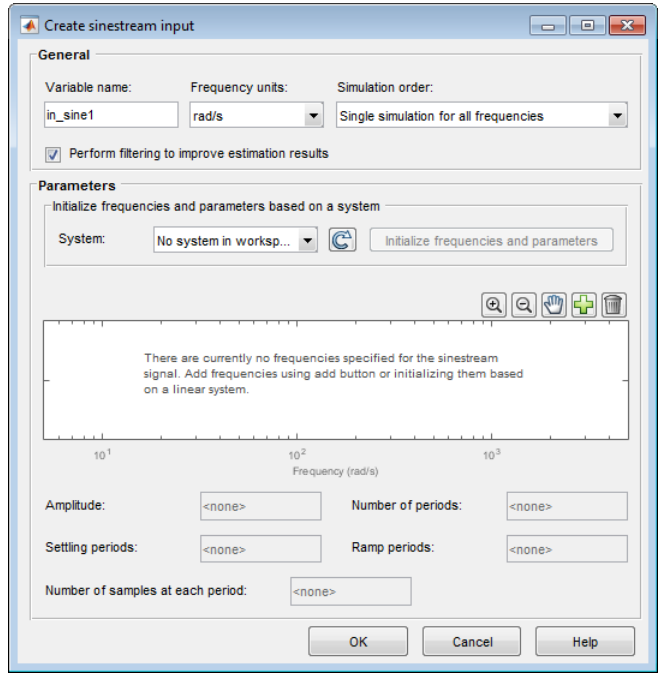
```
sys = 'scdDCMotor';
open_system(sys)
```

2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**. This action opens the Linear Analysis Tool for the model.

Create an input signal for estimation.

1 In the Linear Analysis Tool, click the **Estimation** tab.

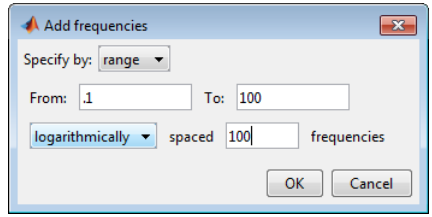
2 In the **Input Signal** drop-down list, select `Sinestream` to open the Create sinestream input dialog box.



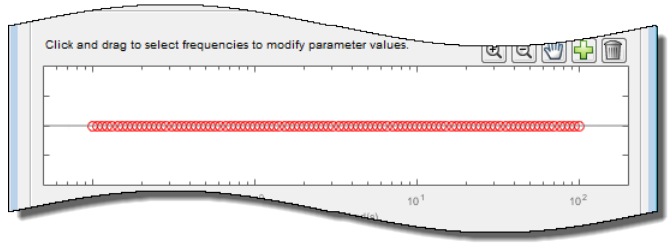
3 Click  to open the Add frequencies dialog box. You can use this dialog box to add frequency points to the input signal.

4 Specify the frequency range for the input.

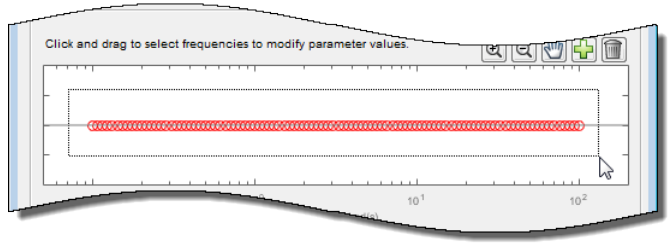
Enter `.1` in the **From** box and `100` in the **To** box. Enter `100` in the box for the number of frequency points.



Click **OK**. This action adds logarithmically spaced frequency points between 0.1 rad/s and 100 rad/s to the input signal. The added points are visible in the Frequency content viewer of the Create sinestream input dialog box.



5 In the Frequency content viewer of the Create sinestream input dialog box, select all the frequency points.



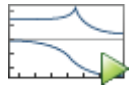
6 Specify the amplitude of the input signal.

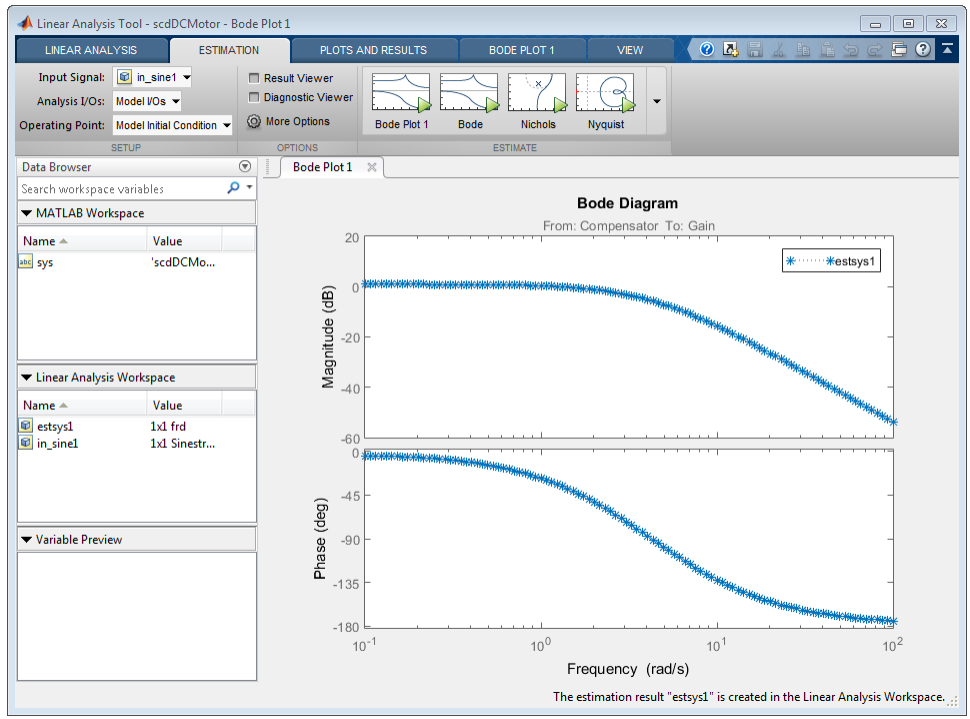
Enter `1` in the **Amplitude** box.

7 Click **OK** to create the sinestream input signal.

The new input signal, `in_sine1`, appears in the **Linear Analysis Workspace**.

Estimate frequency response.

Click  **Bode** to estimate the frequency response. The frequency response estimation result, `estsys1`, appears in the **Linear Analysis Workspace**.



Estimate Frequency Response with Linearization-Based Input Using Linear Analysis Tool

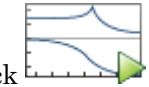
This example shows how to perform frequency response estimation for a model using the Linear Analysis Tool. The input signal used for estimation is based on the linearization of the model.

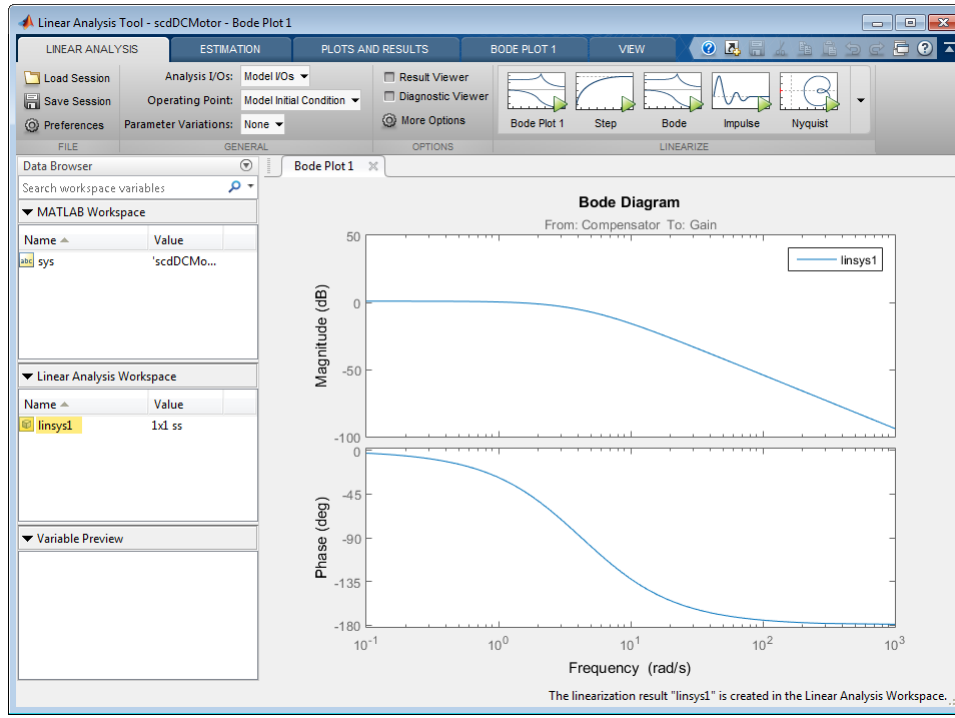
Linearize Simulink model.

1. Open Simulink model.

```
sys = 'scdDCMotor';
open_system(sys)
```

2. In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.
3. Linearize the model using the model initial conditions as the operating point.

   In the **Linear Analysis** tab, click  **Bode**.

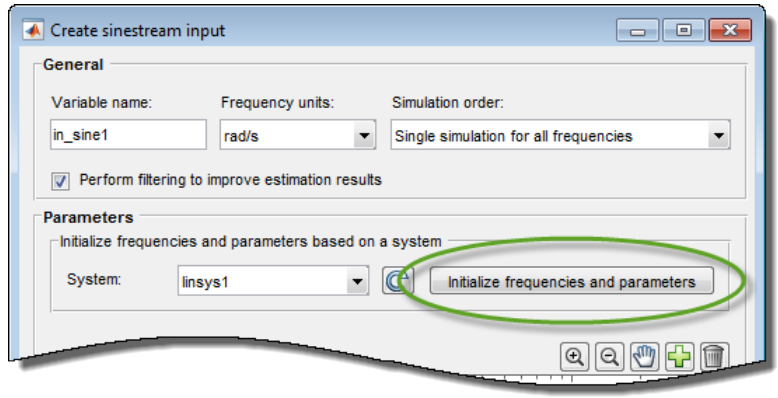


A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

Create sinestream input signal.

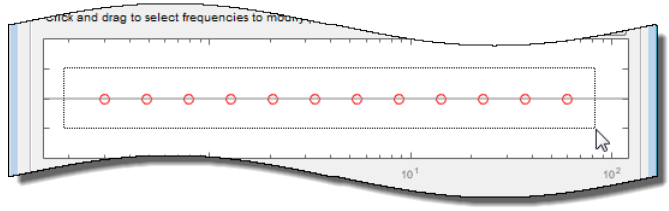
1. In the **Estimation** tab, in the **Input Signal** drop-down list, select `Sinestream`.
2. Initialize the input signal frequencies and parameters based on `linsys1`.

   In the Create sinestream input dialog box, click **Initialize frequencies and parameters**.



The Frequency content viewer is populated with frequency points. The software chooses the frequencies and input signal parameters automatically based on the dynamics of `linsys1`.

3 In the Frequency content viewer, select all the frequency points.



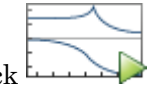
4 Specify the amplitude of the input signal.

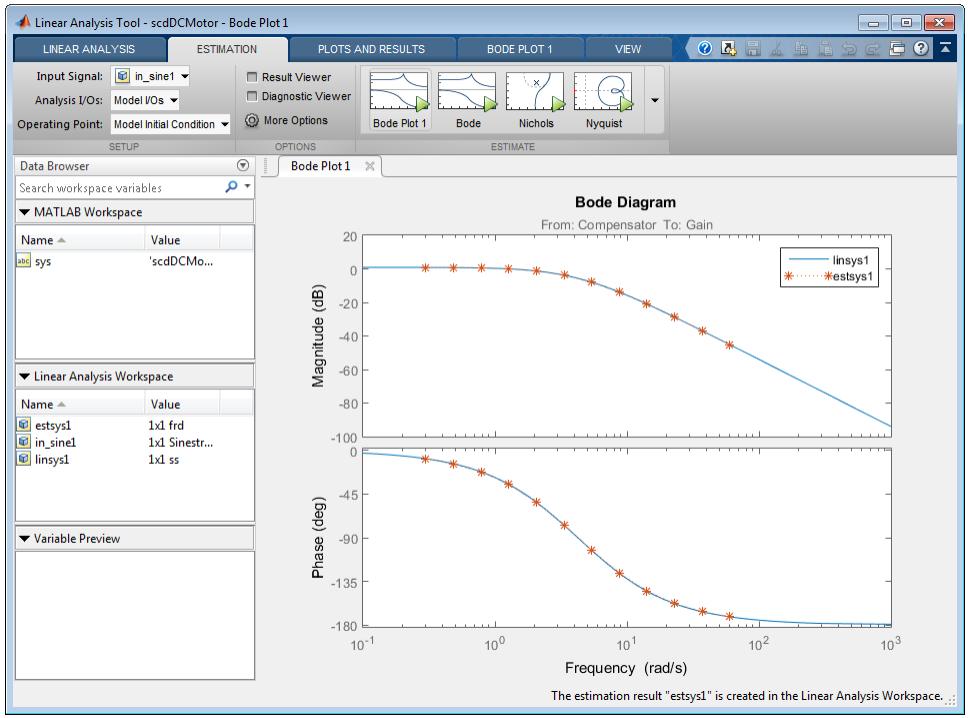
Enter `1` in the **Amplitude** box.

5 Create the input sinestream signal.

Click **OK**. The input signal `in_sine1` appears in the **Linear Analysis Workspace**.

Estimate the frequency response.

Click  **Bode Plot 1** to estimate the frequency response.



The estimated system, `estsys1`, appears in the **Linear Analysis Workspace** and the its frequency response is added to **Bode Plot 1**.

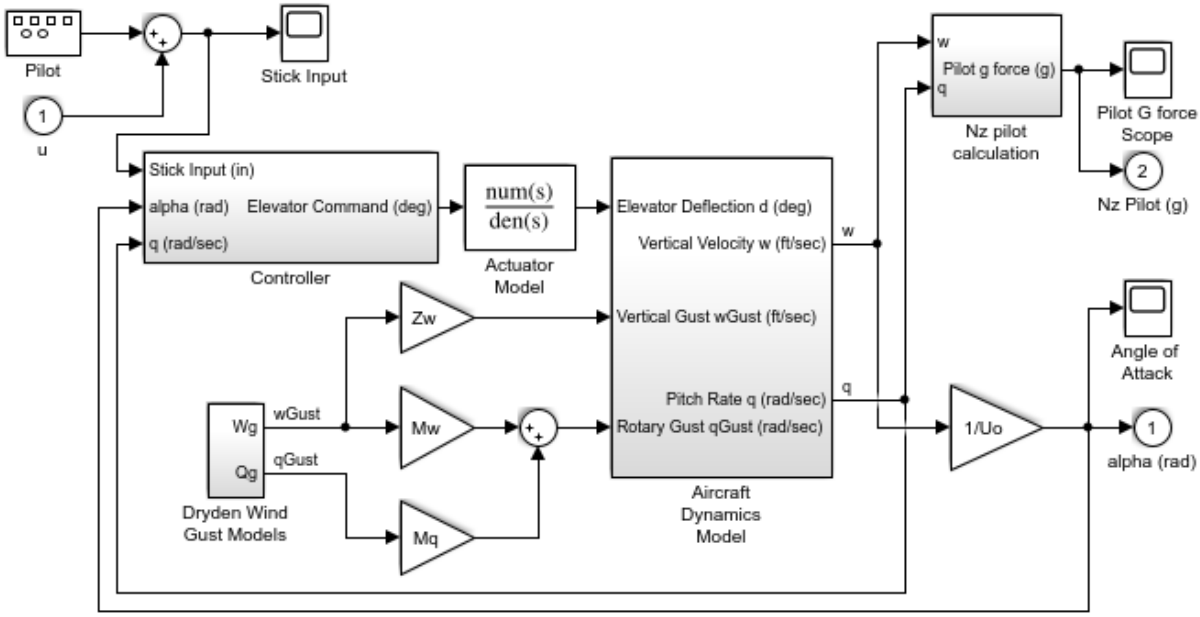
The frequency response for the estimated model matches that of the linearized model.

Estimate Frequency Response at the Command Line

This example shows how to estimate the frequency response of a Simulink® model at the MATLAB® command line.

Open the Simulink model.

```
mdl = 'scdplane';
open_system(mdl)
```



For more information on the general model requirements for frequency response estimation, see “Model Requirements” on page 5-4.

Specify input and output points for frequency response estimation using analysis points. Avoid placing analysis points on bus signals.

```
io(1) = linio('scdplane/Sum1',1);
io(2) = linio('scdplane/Gain5',1,'output');
```

For more information about linear analysis points, see “Specify Portion of Model to Linearize” on page 2-13 and `linio`.

Linearize the model and create a sinestream signal based on the dynamics of the resulting linear system. For more information, see “Estimation Input Signals” on page 5-7 and `frest.Sinestream`.

```
sys = linearize('scdplane',io);
input = frest.Sinestream(sys);
```

If your model has not reached steady state, initialize the model using a steady-state operating point before estimating the frequency response. You can check whether your model is at steady state by simulating the model. For more information on finding steady-state operating points, see “Compute Steady-State Operating Points” on page 1-6.

Find all source blocks in the signal paths of the linearization outputs that generate time-varying signals. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results.

```
srcblks = frest.findSources('scdplane',io);
```

To disable the time-varying source blocks, create an `frestimateOptions` option set and specify the `BlocksToHoldConstant` option.

```
opts = frestimateOptions;
opts.BlocksToHoldConstant = srcblks;
```

Estimate the frequency response.

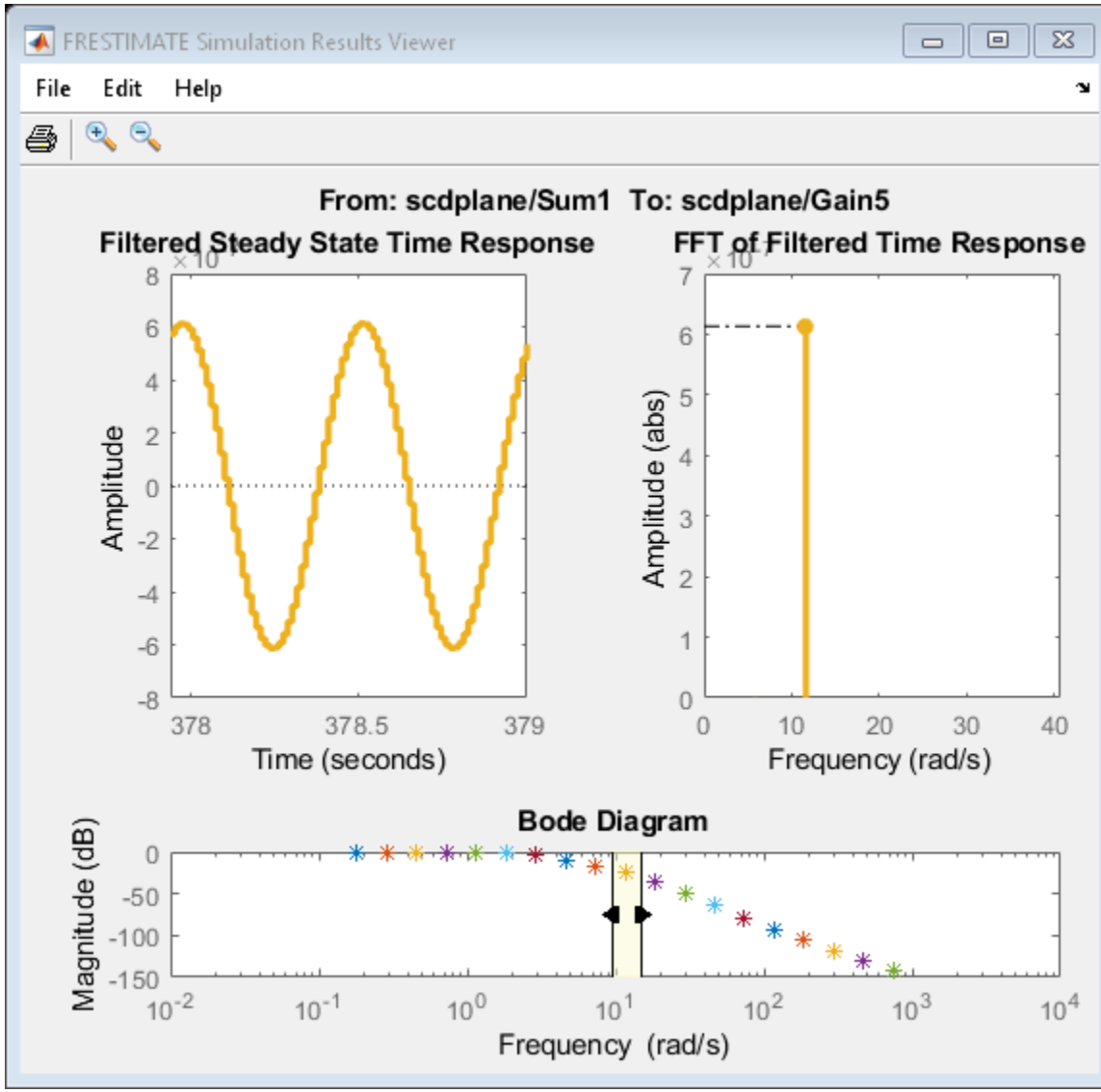
```
[sysest,simout] = frestimate('scdplane',io,input,opts);
```

`sysest` is the estimated frequency response. `simout` is a `Simulink.Timeseries` object representing the simulated output.

To speed up your estimation or decrease its memory requirements, see “Managing Estimation Speed and Memory” on page 5-78.

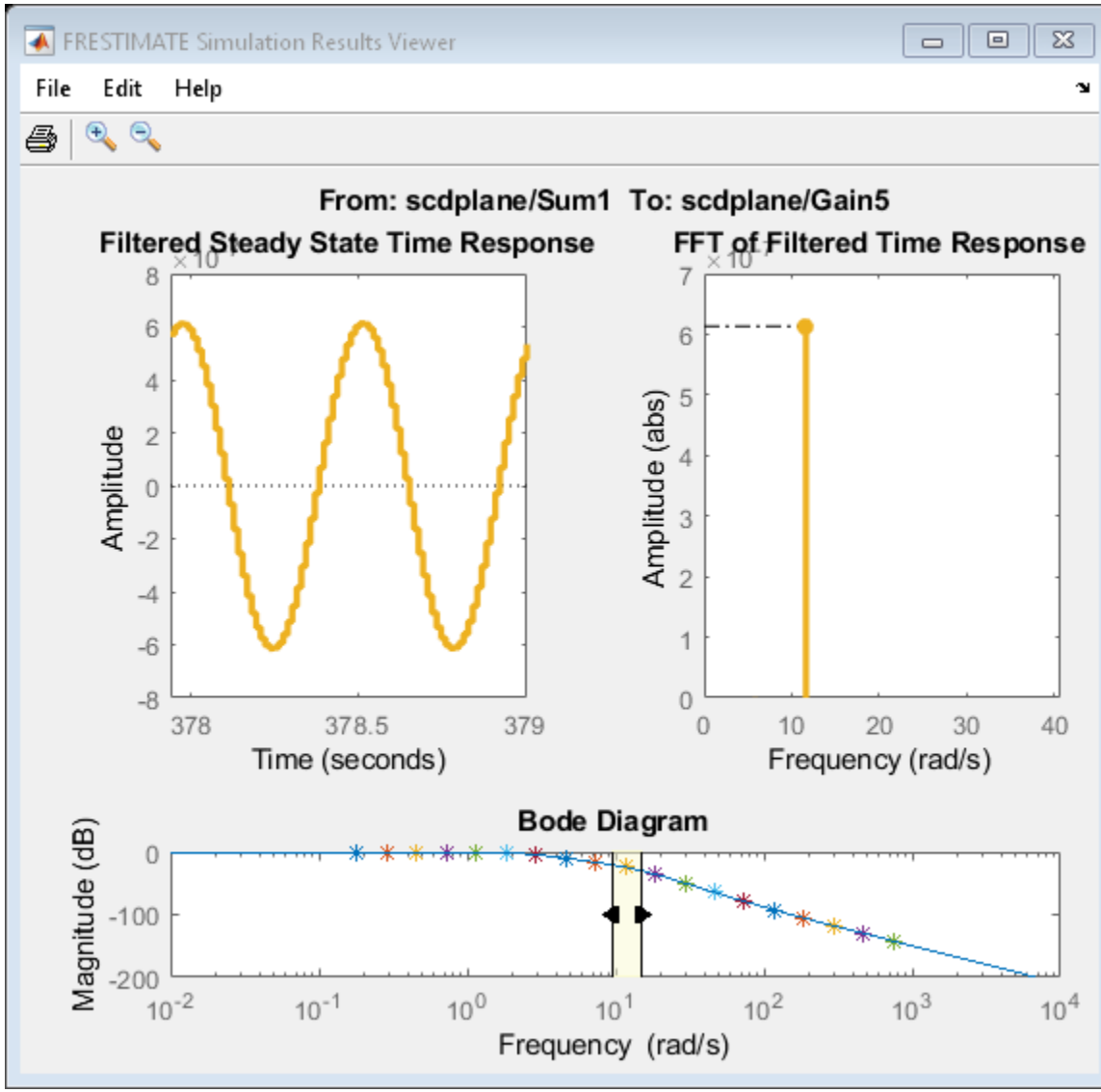
Open the Simulation Results Viewer to analyze the estimated frequency response.

```
frest.simView(simout,input,sysest)
```



You can also compare the estimated frequency response, `sysest`, to an exact linearization of your system, `sys`.

```
frest.simView(simout,input,sysest,sys)
```


FRESTIMATE Simulation Results Viewer
File
Edit
Help
From: scdplane/Sum1 To: scdplane/Gain5
Filtered Steady State Time Response
×10
8
6
4
2
0
-2
-4
-6
-8
Amplitude
378
378.5
379
Time (seconds)
FFT of Filtered Time Response
×10
7
6
5
4
3
2
1
0
Amplitude (abs)
0
10
20
30
40
Frequency (rad/s)
Bode Diagram
Magnitude (dB)
0
-100
-200
10-2
10-1
100
101
102
103
104
Frequency (rad/s)

The **Bode Diagram** plot shows the response `sys` as a blue line.

See Also

`findop` | `frest.findSources` | `frestimate` | `frestimateOptions` | `linio` | `operspec`

More About

- “Estimation Input Signals” on page 5-7
- “Analyze Estimated Frequency Response” on page 5-38

Analyze Estimated Frequency Response

In this section...

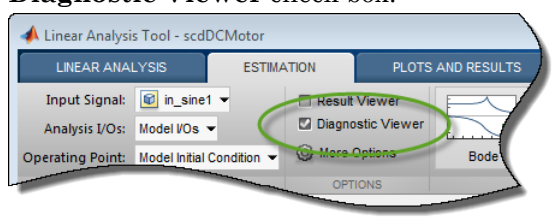
View Simulation Results

View Simulation Results Using Linear Analysis Tool

Use the Diagnostic Viewer to analyze the results of your frequency response estimation, obtained by performing the steps in “Estimate Frequency Response Using Linear Analysis Tool” on page 5-26.

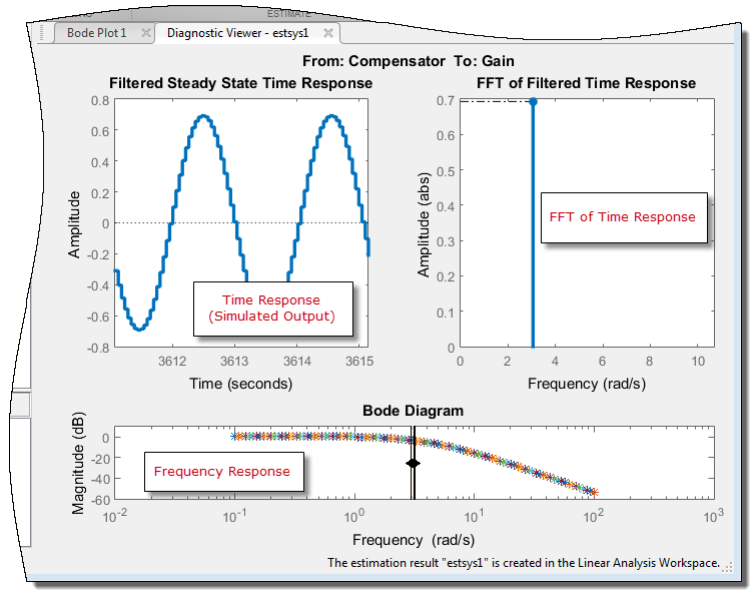
To open the Diagnostic Viewer when estimating a model in the Linear Analysis Tool:

1 Before performing the estimation task, in the **Estimation** tab, select the **Diagnostic Viewer** check box.



This action sets the Diagnostic Viewer to open when the frequency response estimation is performed.

2 In the **Estimate** section, select your desired plot option to estimate the frequency response of the model. The Diagnostic Viewer appears in the plot pane.



To open the Diagnostic Viewer to view a previously estimated model in the Linear Analysis Tool:

1 In the **Linear Analysis Workspace**, select the estimated model.

2 In the **Plots and Results** tab, select the  **Diagnostic Viewer**.

Note This option is only available for models that have been previously estimated with the **Diagnostic Viewer** check box selected.

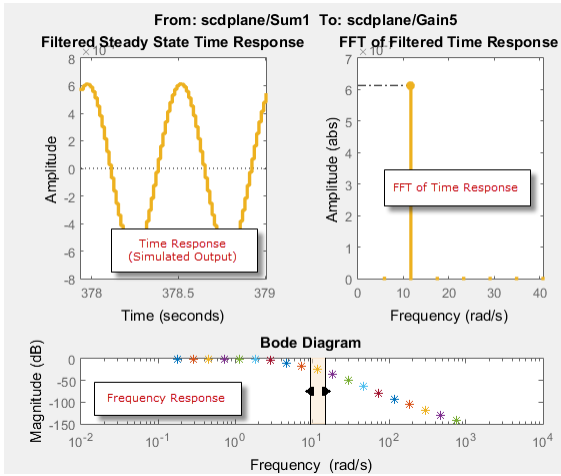
View Simulation Results (MATLAB Code)

Use the Simulation Results Viewer to analyze the results of your frequency response estimation, obtained by performing the steps in “Estimate Frequency Response at the Command Line” on page 5-33.

Open the Simulation Results Viewer using the `frest.simView` command:

```
frest.simView(simout,input,sysest)
```

where `simout` is the simulated output, `input` is the input signal you created, and `sysest` is the estimated frequency response.



Interpret Frequency Response Estimation Results

Select Plots Displayed in Diagnostic Viewer

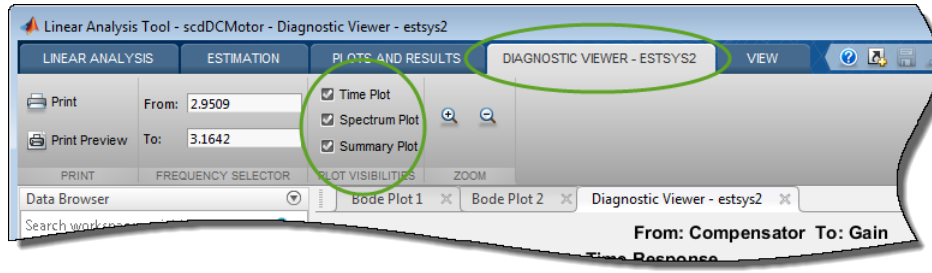
By default, the Diagnostic Viewer shows these plots:

- Frequency Response
- Time Response (Simulated Output)
- FFT of Time Response

To select the plots displayed in the Diagnostic Viewer using the Linear Analysis Tool:

1 If the **Diagnostic Viewer** tab is not visible, in the **Plots and Results** tab, select the **Diagnostic Viewer** plot.

2 In the **Diagnostic Viewer** tab, in the **Plot Visibilities** section, select the check boxes for the plots that you want to view.



To modify plot settings, such as axis frequency units, right-click on a plot, and select the corresponding option.

Select Plots Displayed in Simulation Results Viewer

By default, the Simulation Results Viewer shows these plots:

- Frequency Response
- Time Response (Simulated Output)
- FFT of Time Response

To select the plots displayed in the Simulation Results Viewer, choose the corresponding plot from the **Edit** > **Plots** menu. To modify plot settings, such as axis frequency units, right-click a plot, and select the corresponding option.

Frequency Response

Use the Bode plot to analyze the frequency response. If the frequency response does not match the dynamics of your system, see “Troubleshooting Frequency Response Estimation” on page 5-46 for information about possible causes and solutions. While troubleshooting, you can use the Bode plot controls to view the time response at the problematic frequencies on page 5-42.

You can usually improve estimation results by either modifying your input signal on page 5-23 or disabling the model blocks that drive your system away from the operating point, and repeating the estimation.

Time Response (Simulated Output)

Use this plot to check whether the simulated output is at steady state at specific frequencies. If the response has not reached steady state, see “Time Response Not at Steady State” on page 5-46 for possible causes and solutions.

If you used the sinestream input for estimation, check both the filtered and the unfiltered time response. You can toggle the display of filtered and unfiltered output by right-clicking the plot and selecting **Show filtered steady state output only**. If both the filtered and unfiltered response appear at steady state, then your model must be at steady state. You can explore other possible causes in “Troubleshooting Frequency Response Estimation” on page 5-46.

Note If you used the sinestream input for estimation, toggling the filtered and unfiltered display only updates the Time Response and FFT plots. This selection does not change the estimation results. For more information about filtering during estimation, see “How Frequency Response Estimation Treats Sinestream Inputs” on page 5-8.

FFT of Time Response

Use this plot to analyze the spectrum of the simulated output.

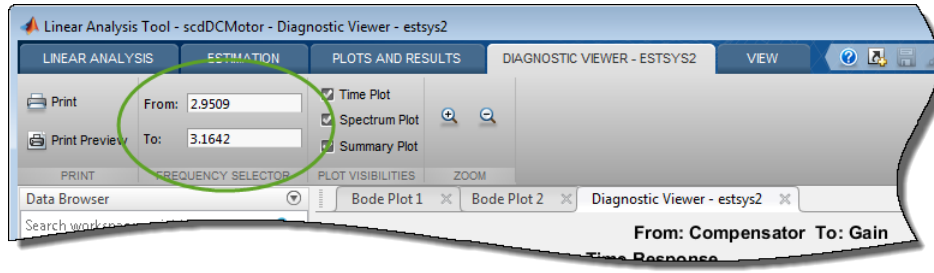
For example, you can use the spectrum to identify strong nonlinearities. When the FFT plot shows large amplitudes at frequencies other than the input signal, your model is operating outside of linear range. If you are interested in analyzing the linear response of your system for small perturbations, explore possible solutions in “FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency” on page 5-50.

Analyze Simulated Output and FFT at Specific Frequencies

Using the Diagnostic Viewer in Linear Analysis Tool

Use the controls in the **Diagnostic Viewer** tab of the Linear Analysis Tool to analyze the estimation results at specific frequencies.

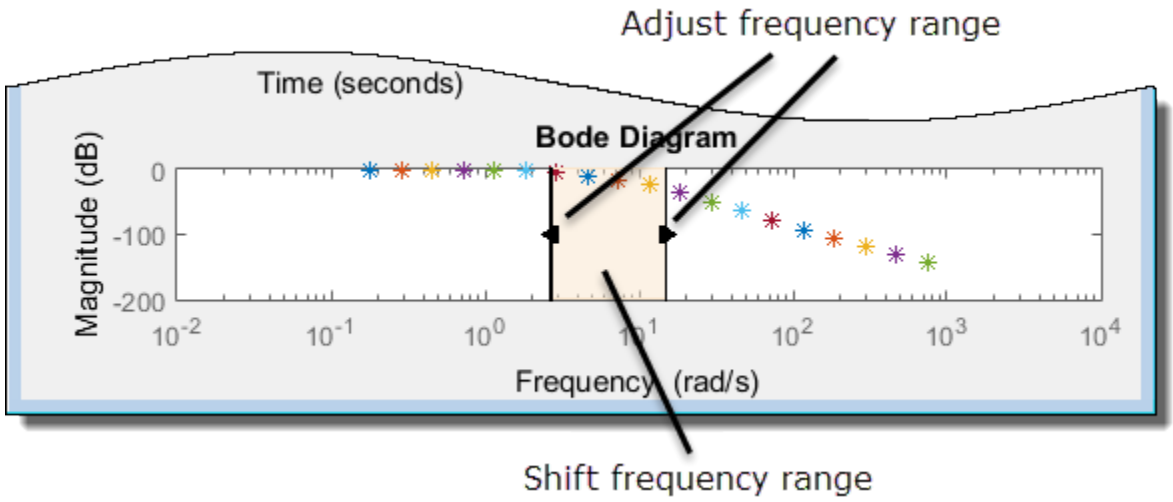
1. If the **Diagnostic Viewer** tab is not visible, in the **Plots and Results** tab, select the **Diagnostic Viewer** plot.
2. In the **Diagnostic Viewer** tab, in the **Frequency Selector** section, specify the frequency range that you want to inspect. Use the frequency units used in the Bode plot in the Diagnostic Viewer.



Using the Simulation Results Viewer

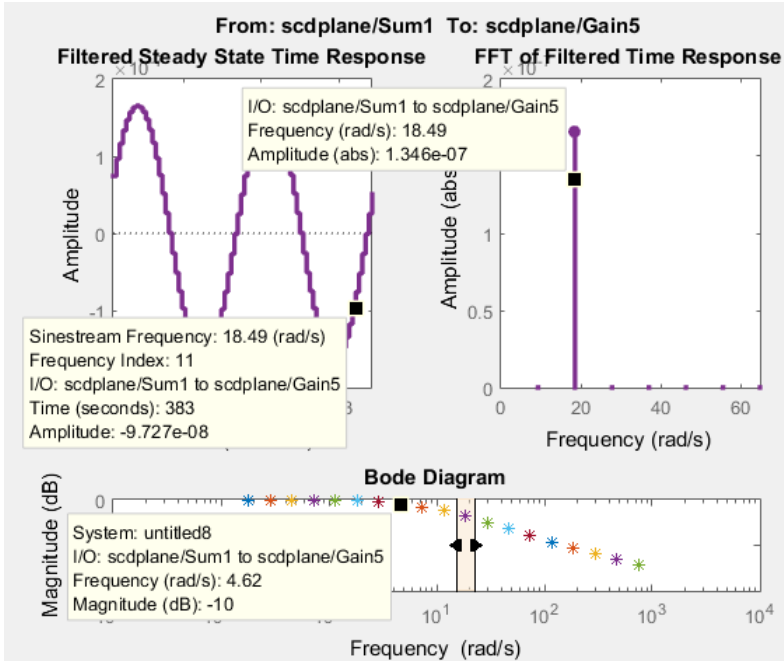
In the Simulation Results Viewer, use the Bode controls to display the simulated output and its spectrum at specific frequencies.

- Drag the arrows individually to display the time response and FFT at specific frequencies.
- Drag the shaded region to shift the time response and FFT to a different frequency range.



Annotate Frequency Response Estimation Plots

You can display a data tip on the Time Response, FFT, and Bode plots in the Simulation Results Viewer by clicking the corresponding curve. Dragging the data tip updates the information.



Data tips are useful for correcting poor estimation results at a specific sinestream frequency, which requires you to modify the input at a specific frequency. You can use the data tip to identify the frequency index where the response does not match your system.

In the previous figure, the Time Response data tip shows that the frequency index is 11. You can use this frequency index to modify the corresponding portion of the input signal. For example, to modify the `NumPeriods` and `SettlingPeriods` properties of the sinestream signal, using MATLAB code:

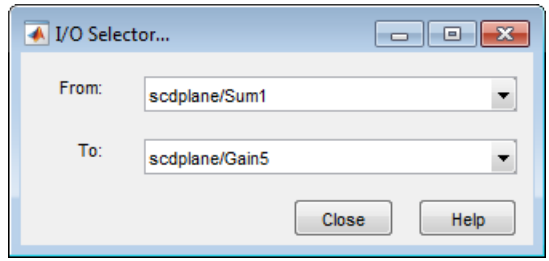
```
input.NumPeriods(11) = 80;
input.SettlingPeriods(11) = 75;
```

To modify the sinestream in the Linear Analysis Tool, see “Modify Sinestream Signal Using Linear Analysis Tool” on page 5-23

Displaying Estimation Results for Multiple-Input Multiple-Output (MIMO) Systems

For MIMO systems, view frequency response information for specific input and output channels:

1. In both the Diagnostic Viewer and Simulation Results Viewer, right-click any plot, and select **I/O Selector**.
2. Choose the input channel in the **From** list and the output channel in the **To** list.



Troubleshooting Frequency Response Estimation

In this section...

When to Troubleshoot

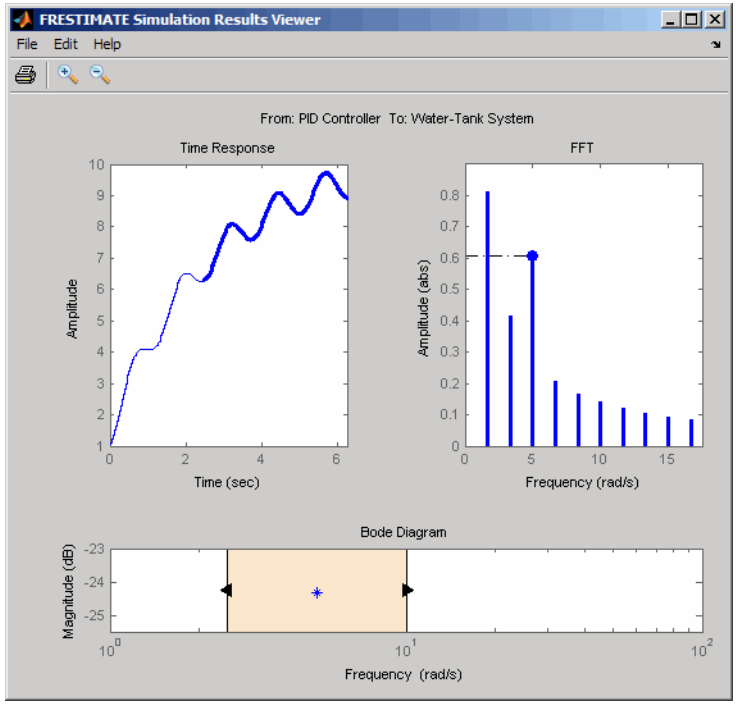
If, after analyzing your frequency response estimation, the frequency response plot does not match the expected behavior of your system, you can use the time response and FFT plots to help you improve the results.

If your estimation is slow or you run out of memory during estimation, see “Managing Estimation Speed and Memory” on page 5-78.

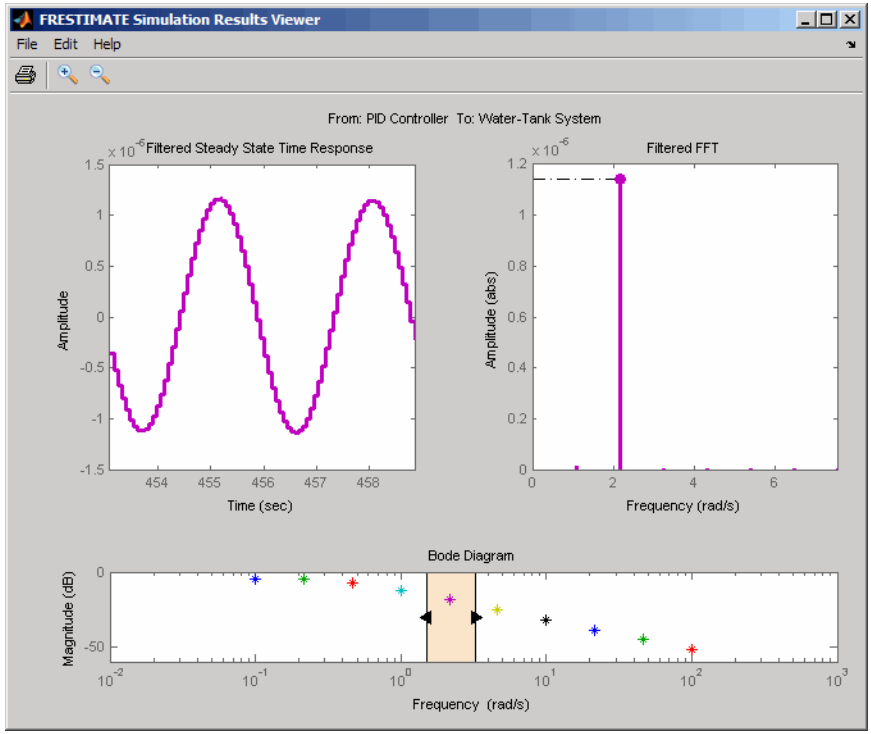
Time Response Not at Steady State

What Does This Mean?

This time response has not reached steady state.



This plot shows a steady-state time response.



Because frequency response estimation requires steady-state input and output signals, transients produce inaccurate estimation results.

For sinestream input signals, transients sometimes interfere with the estimation either directly or indirectly through spectral leakage. For chirp input signals, transients interfere with estimation.

How Do I Fix It?

Possible Cause	Action
Model cannot initialize to steady state.	• Increase the number of periods for frequencies that do not reach steady state by changing the `NumPeriods` and `SettlingPeriods` properties. See "Modify Estimation Input Signals" on page 5-23. • Disable all time-varying source blocks in your model and repeat the estimation. See "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 5-58.
(Sinestream input) Not enough periods for the output to reach steady state.	• Increase the number of periods for frequencies that do not reach steady state by changing the `NumPeriods` and `SettlingPeriods`. See "Modify Estimation Input Signals" on page 5-23. • Check that filtering is enabled during estimation. You enable filtering by setting the `ApplyFilteringInFRESTIMATE` option to `on`. For information about how estimation uses filtering, see the `frestimate` reference page.
(Chirp input) Signal sweeps through the frequency range too quickly.	Increase the simulation time by increasing `NumSamples`. See "Modify Estimation Input Signals" on page 5-23.

After you try the suggested actions, recompute the estimation either:

- At all frequencies
- In a particular frequency range (only for sinestream input signals)

To recompute the estimation in a particular frequency range:

1. Determine the frequencies for which you want to recompute the estimation results. Then, extract a portion of the sinestream input signal at these frequencies using `fselect`.

   For example, these commands extract a sinestream input signal between 10 and 20 rad/s from the input signal `input`:

   ```
   input2 = fselect(input,10,20);
   ```

2. Modify the properties of the extracted sinestream input signal `input2`, as described in "Modify Estimation Input Signals" on page 5-23.
3. Estimate the frequency response `sysest2` with the modified input signal using `frestimate`.
4. Merge the original estimated frequency response `sysest` and the recomputed estimated frequency response `sysest2`:

   a. Remove data from `sysest` at the frequencies in `sysest2` using `fdel`.

   ```
   sysest = fdel(sysest,input2.Frequency)
   ```

   b. Concatenate the original and recomputed responses using `fcat`.

   ```
   sys_combined = fcat(sysest2,sysest)
   ```

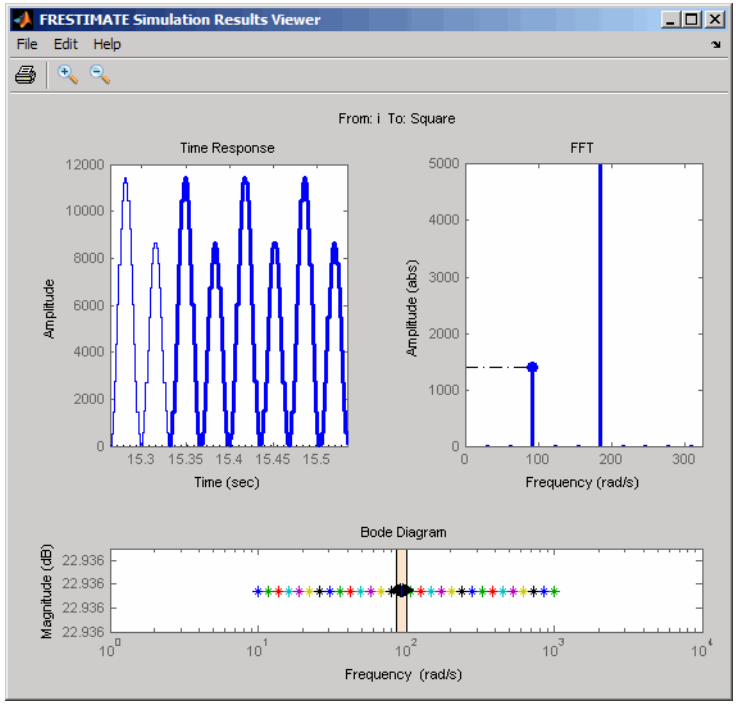
Analyze the recomputed frequency response, as described in "Analyze Estimated Frequency Response" on page 5-38.

For an example of frequency response estimation with time-varying source blocks, see "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 5-58

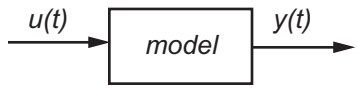
FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency

What Does This Mean?

When the FFT plot shows large amplitudes at frequencies other than the input signal, your model is operating outside the linear range. This condition can cause problems when you want to analyze the response of your linear system to small perturbations.



For models operating in the linear range, the input amplitude A_1 in $y(t)$ must be larger than the amplitudes of other harmonics, A_2 and A_3.



$$u(t) = A_1 \sin(\omega_1 + \phi_1)$$

$$y(t) = A_1 \sin(\omega_1 + \phi_1) + A_2 \sin(\omega_2 + \phi_2) + A_3 \sin(\omega_3 + \phi_3) + \ldots$$

How Do I Fix It?

Adjust the amplitude of your input signal to decrease the impact of other harmonics, and repeat the estimation. Typically, you should decrease the input amplitude level to keep the model operating in the linear range.

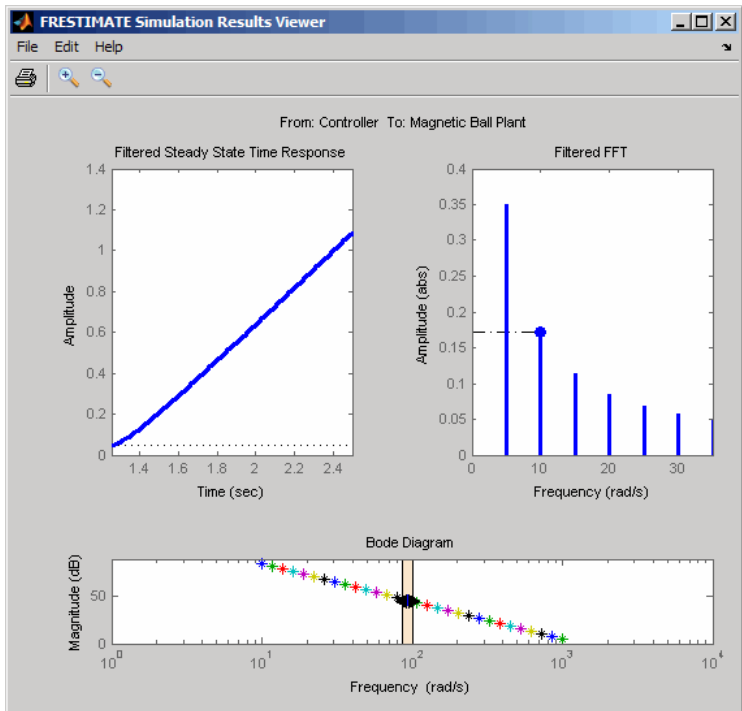
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- “Modify Estimation Input Signals” on page 5-23

Time Response Grows Without Bound

What Does This Mean?

When the time response grows without bound, frequency response estimation results are inaccurate. Frequency response estimation is only accurate close to the operating point.



How Do I Fix It?

Try the suggested actions listed the table and repeat the estimation.

Possible Cause	Action
Model is unstable.	You cannot estimate the frequency response using `frestimate`. Instead, use exact linearization to get a linear representation of your model. See "Linearize Simulink Model at Model Operating Point" on page 2-69 or the `linearize` reference page.
Stable model is not at steady state.	Disable all source blocks in your model, and repeat the estimation using a steady-state operating point. See "Compute Steady-State Operating Points" on page 1-6.
Stable model captures a growing transient.	If the model captures a growing transient, increase the number of periods in the input signal by changing `NumPeriods`. Repeat the estimation using a steady-state operating point.

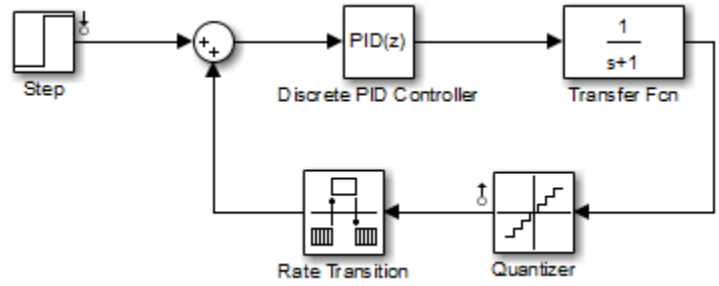
Time Response Is Discontinuous or Zero

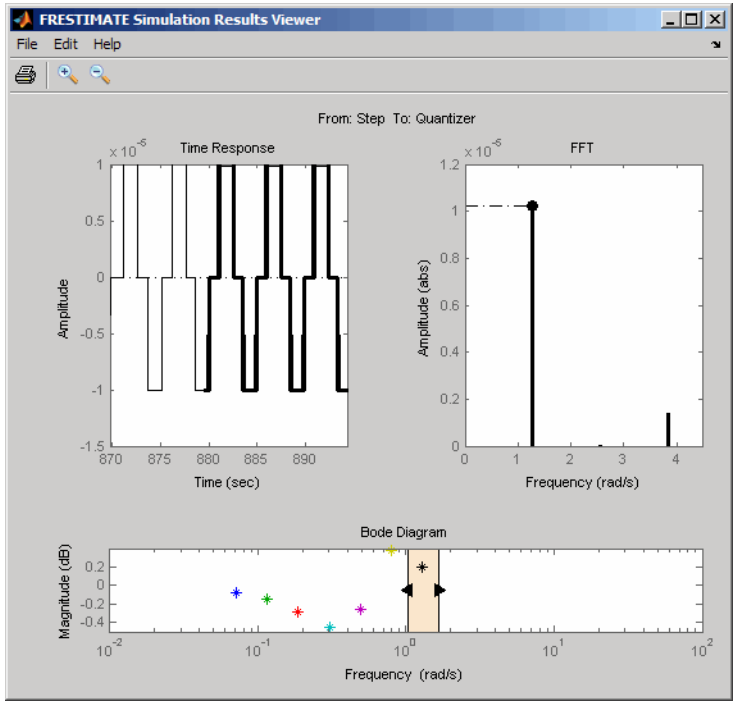
What Does This Mean?

Discontinuities or noise in the time response indicate that the amplitude of your input signal is too small to overcome the effects of the discontinuous blocks in your model. Examples of discontinuous blocks include Quantizer, Backlash, and Dead Zones.

If you used a sinestream input signal and estimated with filtering, turn filtering off in the Simulation Results Viewer to see the unfiltered time response.

The following model with a Quantizer block shows an example of the impact of an input signal that is too small. When you estimate this model, the unfiltered simulation output includes discontinuities.

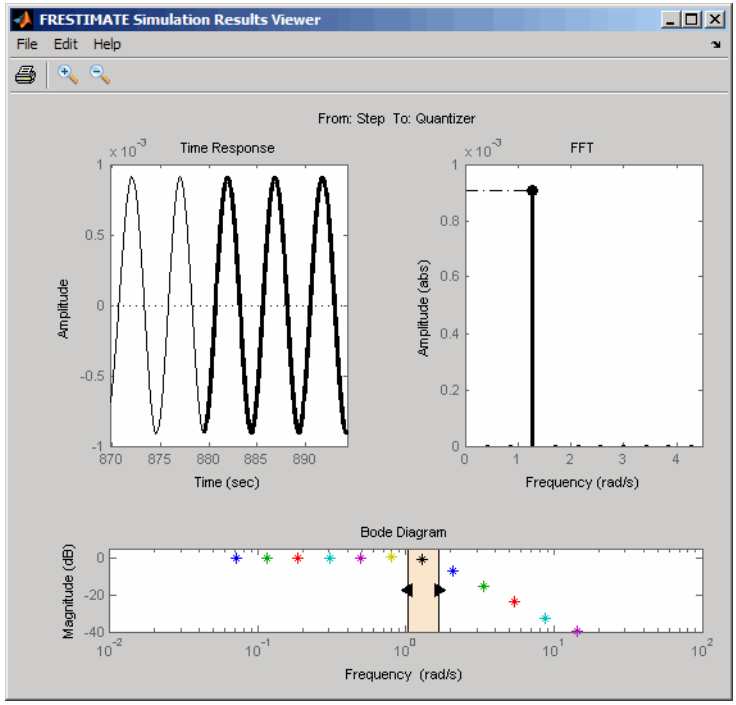




How Do I Fix It?

Increase the amplitude of your input signal, and repeat the estimation.

With a larger amplitude, the unfiltered simulated output of the model with a Quantizer block is smooth.



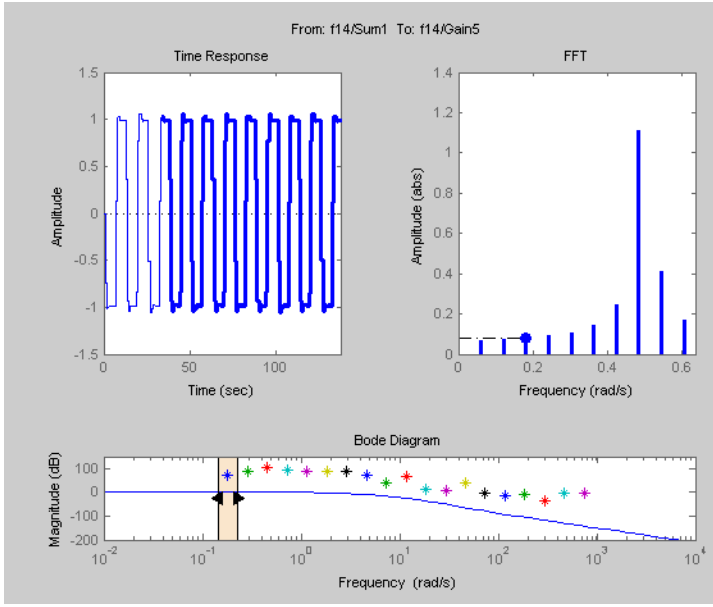
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- “Modify Estimation Input Signals” on page 5-23

Time Response Is Noisy

What Does This Mean?

When the time response is noisy, frequency response estimation results may be biased.



How Do I Fix It?

`frestimate` does not support estimating frequency response estimation of Simulink models with blocks that model noise. Locate such blocks with `frest.findSources` and disable them using the `BlocksToHoldConstant` option of `frestimate`.

If you need to estimate a model with noise, use `frestimate` to simulate an output signal from your Simulink model for estimation—without modifying your model. Then, use the Signal Processing Toolbox™ or System Identification Toolbox software to estimate a model.

To simulate the output of your model in response to a specified input signal:

1 Create a random input signal. For example:

```
in = frest.Random('Ts',0.001,'NumSamples',1e4);
```

You can also specify your own custom signal as a `timeseries` object. For example:

```
t = 0:0.001:10;
y = sin(2*pi*t);
in_ts = timeseries(y,t);
```

2 Simulate the model to obtain the output signal. For example:

```
[sysest,simout] = frestimate(model,op,io,in_ts)
```

The second output argument of `frestimate`, `simout`, is a `Simulink.Timeseries` object that stores the simulated output. `in_ts` is the corresponding input data.

3 Generate `timeseries` objects before using with other MathWorks® products:

```
input = generateTimeseries(in_ts);
output = simout{1}.Data;
```

You can use data from `timeseries` objects directly in Signal Processing Toolbox software, or convert these objects to System Identification Toolbox data format. For examples, see "Estimate Frequency Response Models with Noise Using Signal Processing Toolbox" on page 5-73 and "Estimate Frequency Response Models with Noise Using System Identification Toolbox" on page 5-75.

For a related example, see "Disable Noise Sources During Frequency Response Estimation" on page 5-67.

Effects of Time-Varying Source Blocks on Frequency Response Estimation

Setting Time-Varying Sources to Constant for Estimation Using Linear Analysis Tool

This example illustrates the effects of time-varying sources on estimation. The example also shows how to set time-varying sources to be constant during estimation to improve estimation results.

1 Open the Simulink model.

```
sys = 'scdspeed_ctrlloop';
open_system(sys)
```

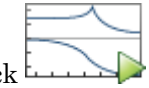
2 Linearize the model.

a Set the `Engine Model` block to normal mode for accurate linearization.

```
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')
```

b Open the Linear Analysis Tool for the model.

In the Simulink Editor, select **Analysis** > **Control Design** > **Linear Analysis**.

c Click  **Bode** to linearize the model and generate a Bode plot of the result.

The linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

3 Create an input sinestream signal for the estimation.

a Open the Create sinestream input dialog box.

In the **Estimation** tab, in the **Input Signal** drop-down list, select **Sinestream**.

b Open the Add frequencies dialog box.

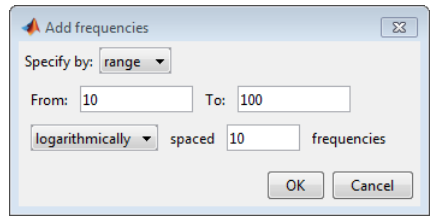
Click  .

c Specify the input sinestream frequency range and number of frequency points.

Enter `10` in the **From** box.

Enter `100` in the **To** box.

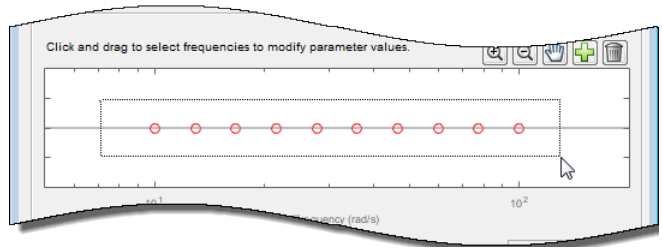
Enter `10` in the box for the number of frequency points.



Click **OK**.

The added points are visible in the Frequency content viewer of the Create sinestream input dialog box.

d In the Frequency content viewer of the Create sinestream input dialog box, select all the frequency points.



e Specify input sinestream parameters.

Change the **Number of periods** and **Settling periods** to ensure that the model reaches steady-state for each frequency point in the input sinestream.

Enter `30` in the **Number of periods** box.

Enter `25` in the **Settling periods** box.

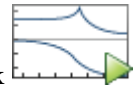
f Create the input sinestream.

Click **OK**. The new input signal, `in_sine1`, appears in the **Linear Analysis Workspace**.

4 Set the Diagnostic Viewer to open when estimation is performed.

Select the **Launch Diagnostic Viewer** check box.

5 Estimate the frequency response for the model.

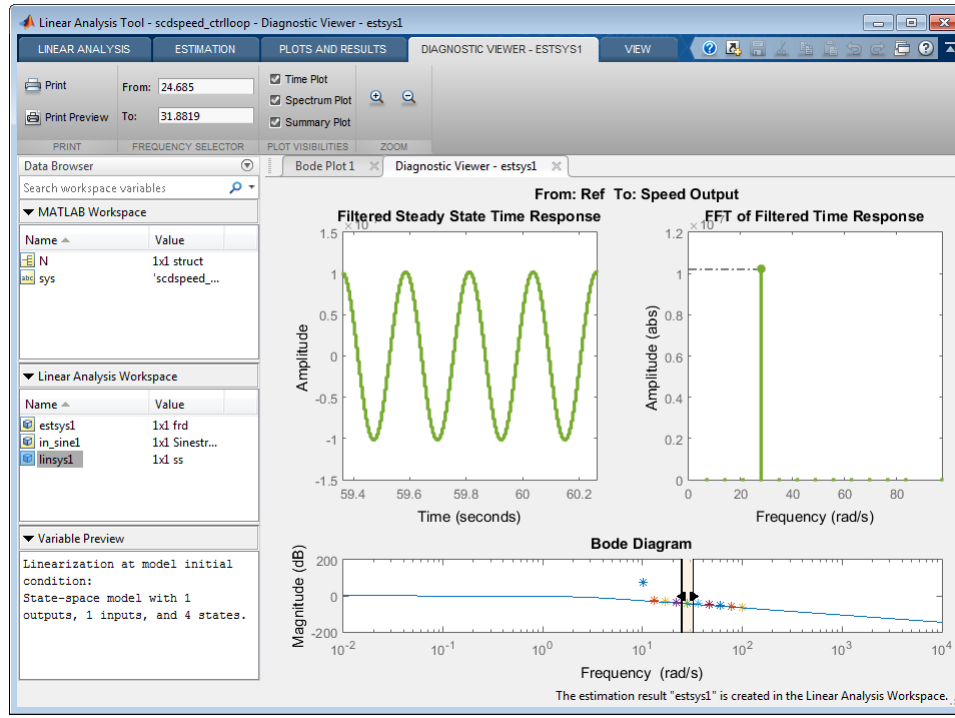
Click  **Bode Plot 1** to estimate the frequency response. The Diagnostic Viewer appears in the plot plane and the estimated system `estsys1`, appears in the **Linear Analysis Workspace**.

6 Compare the estimated model and the linearized model.

a Click on the **Diagnostic Viewer - estsys1** tab in the plot area of the Linear Analysis Tool.

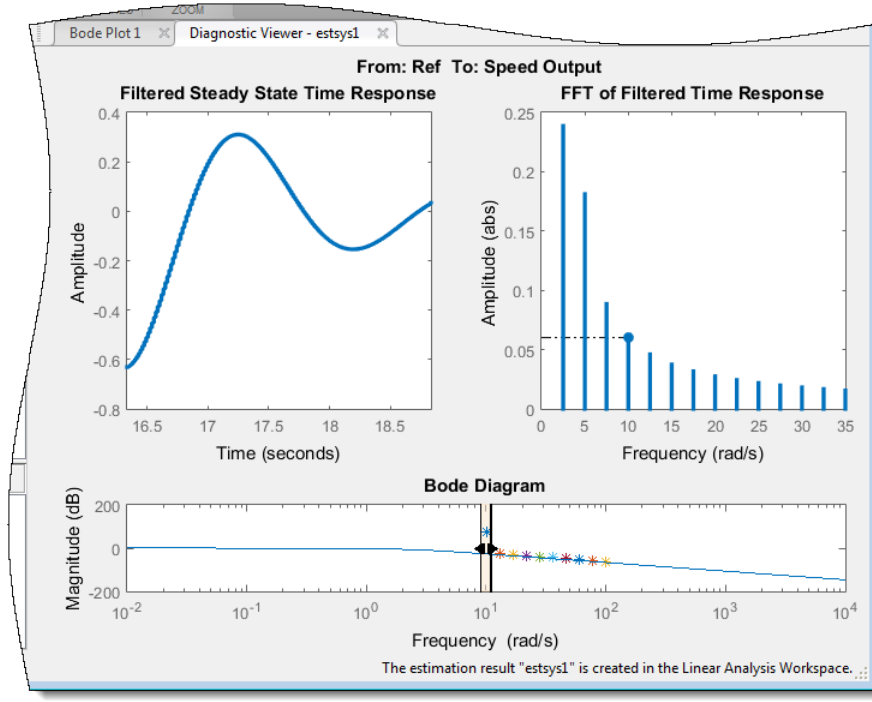
b Click and drag `linsys1` onto the Diagnostic Viewer to add `linsys1` to the **Bode Diagram**.

c Click the **Diagnostic Viewer** tab.



d Configure the Diagnostic Viewer to show only the frequency point where the estimation and linearization results do not match.

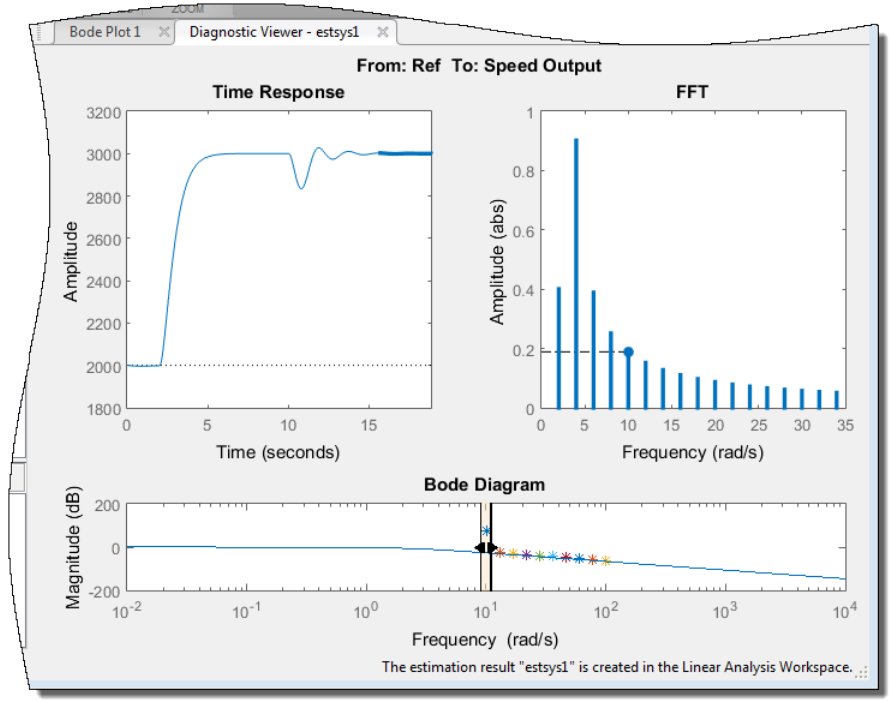
In the **Frequency Selector** section, enter `9` in the **From** box and `11` in the **To** box to set the frequency range that is analyzed in the Diagnostic Viewer.



The **Filtered Steady State Time Response** plot shows a signal that is not sinusoidal.

e View the unfiltered time response.

Right-click the **Filtered Steady State Time Response** plot and clear the **Show filtered steady state output only** option.

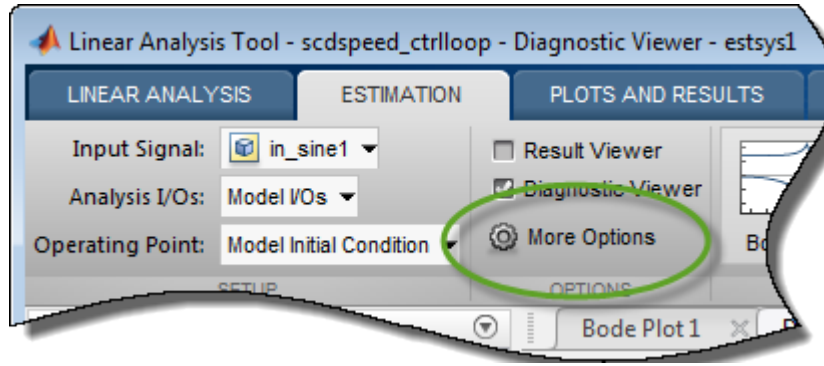


The step input and external disturbances drive the model away from the operating point that the linearized model uses. This prevents the response from reaching steady-state. To correct this problem, find and disable the time-varying source blocks that interfere with the estimation. Then estimate the frequency response of the model again.

7 Find and disable the time-varying sources within the model.

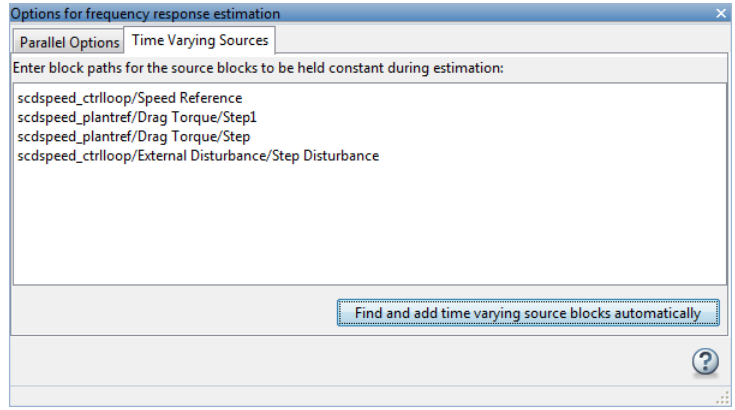
a Open the Options for frequency response estimation dialog box.

In the **Estimation** tab, in the **Options** section, click **More Options**.

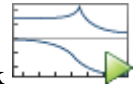


b In the **Time Varying Sources** tab, click **Find and add time varying source blocks automatically**.

This action populates the time varying sources list with the block paths of the time varying sources in the model. These sources will be held constant during estimation.



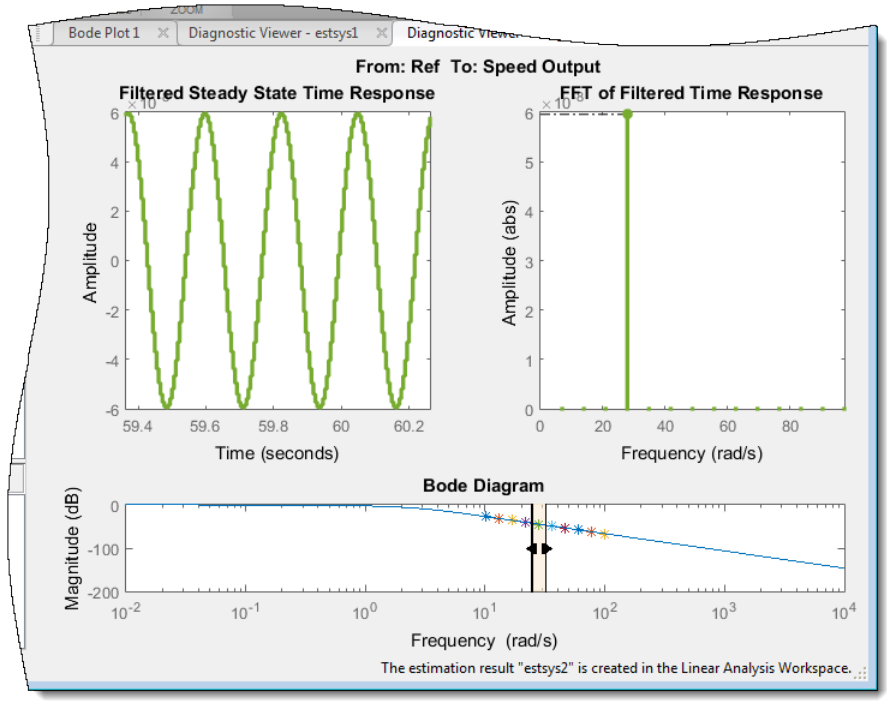
8 Estimate the frequency response for the model.

Click  **Bode Plot 1** to estimate the frequency response. The estimated system `estsys2`, appears in the **Linear Analysis Workspace**.

9 Compare the newly estimated model and the linearized model.

Click on the **Diagnostic Viewer - estsys2** tab in the plot area of the Linear Analysis Tool.

Click and drag `linsys1` onto the Diagnostic Viewer.



The frequency response obtained by holding the time-varying sources constant matches the exact linearization results.

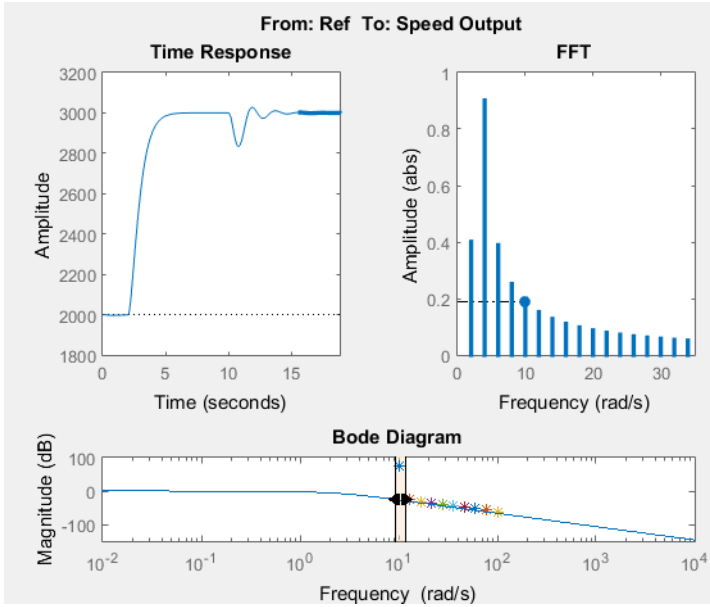
Setting Time-Varying Sources to Constant for Estimation (MATLAB Code)

Compare the linear model obtained using exact linearization techniques with the estimated frequency response:

```
% Open the model
mdl = 'scdspeed_ctrlloop';
open_system(mdl)
io = getlinio(mdl);

% Set the model reference to normal mode for accurate linearization
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')

% Linearize the model
sys = linearize(mdl,io);
```

```
% Estimate the frequency response between 10 and 100 rad/s
in = frest.Sinestream('Frequency',logspace(1,2,10),'NumPeriods',30,'SettlingPeriods',25);
[sysest,simout] = frestimate(mdl,io,in);

% Compare the results
frest.simView(simout,in,sysest,sys)
```

The linearization results do not match the estimated frequency response for the first two frequencies. To view the unfiltered time response, right-click the time response plot, and uncheck **Show filtered steady state output only**.



The step input and external disturbances drive the model away from the operating point, preventing the response from reaching steady-state. To correct this problem, find and disable these time-varying source blocks that interfere with the estimation.

Identify the time-varying source blocks using `frest.findSources`.

```
srcblks = frest.findSources(mdl,io);
```

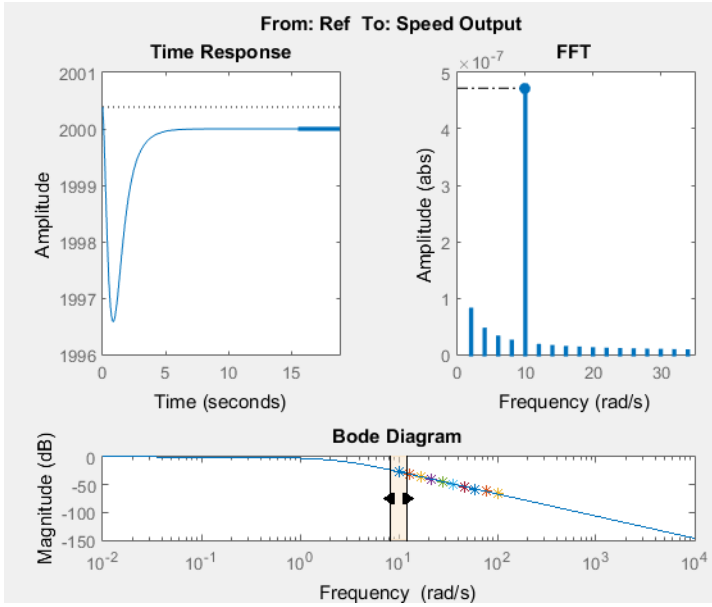
Create a `frestimate` options set to disable the blocks.

```
opts = frestimateOptions;
opts.BlocksToHoldConstant = srcblks;
```

Repeat the frequency response estimation using the optional input argument `opts`.

```
[sysest2,simout2] = frestimate(mdl,io,in,opts);
frest.simView(simout2,in,sysest2,sys)
```

Now the resulting frequency response matches the exact linearization results. To view the unfiltered time response, right-click the time response plot, and uncheck **Show filtered steady state output only**.

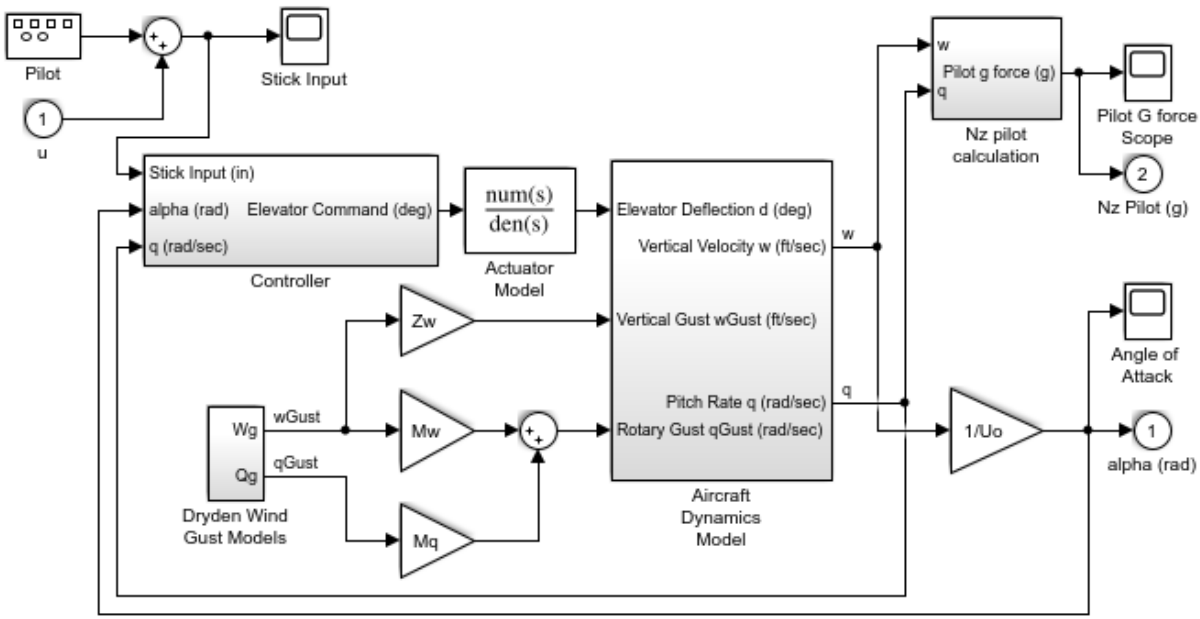


Disable Noise Sources During Frequency Response Estimation

This example shows how to disable noise sources in your Simulink® model during frequency response estimation. Such noise sources can interfere with the signal at the linearization output points and produce inaccurate estimation results.

Open the model.

```
mdl = 'scdplane';
open_system(mdl)
```



Specify linearization input and output points.

```
io(1) = linio('scdplane/Sum1',1);
io(2) = linio('scdplane/Gain5',1,'output');
```

Linearize the model and create a sinestream estimation input signal based on the dynamics of the resulting linear system.

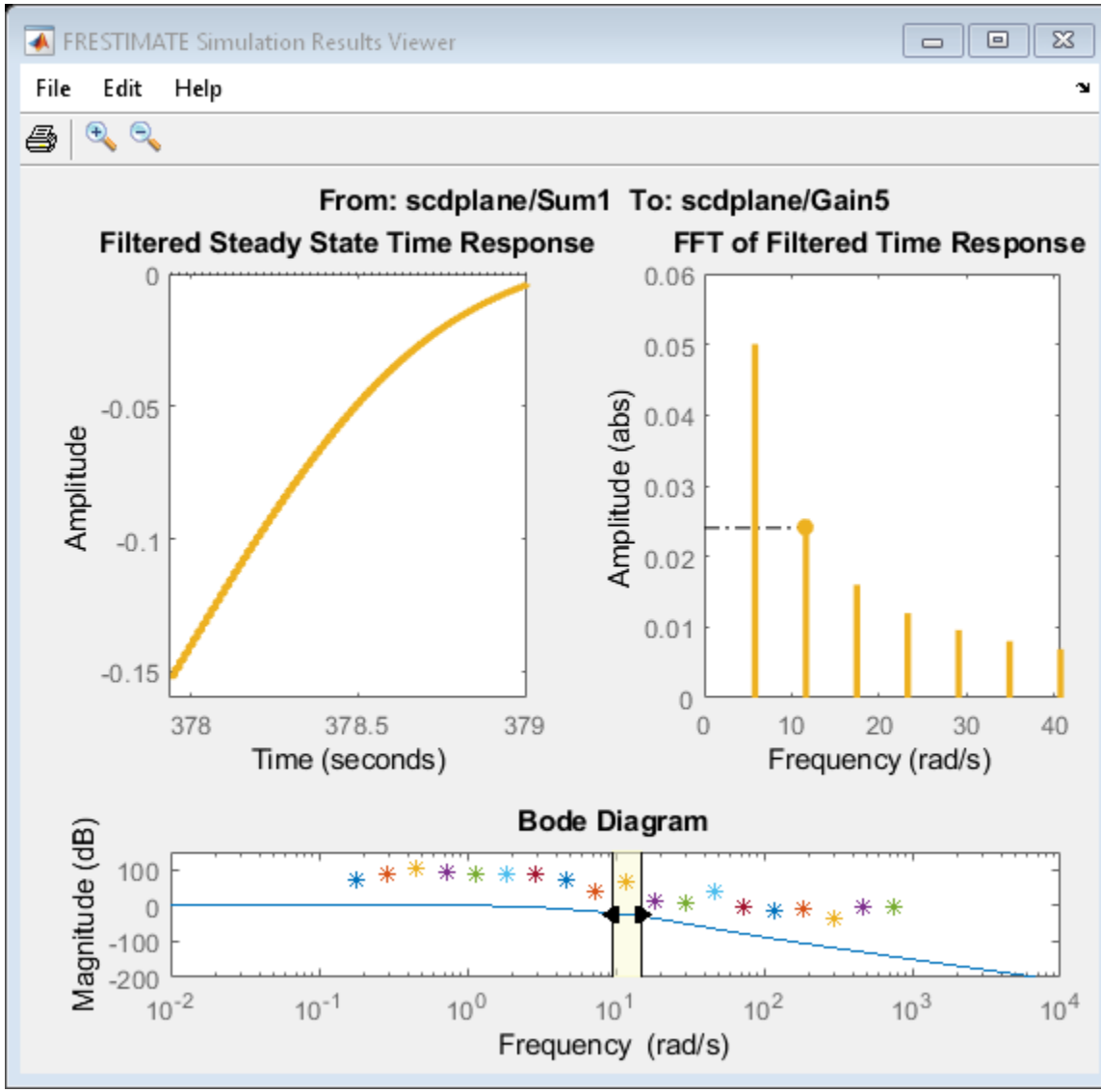
```
sys = linearize(mdl,io);
in = frest.Sinestream(sys);
```

Estimate frequency response.

```
[sysest,simout] = frestimate(mdl,io,in);
```

Compare the estimated frequency response to the exact linearization result.

```
frest.simView(simout,in,sysest,sys)
```



In the **Bode Diagram**, the estimated frequency response does not match the response of the exact linearization. This result is due to the effects of the Pilot and Wind Gust Disturbance blocks in the model. To view the effects of the noise on the time response at

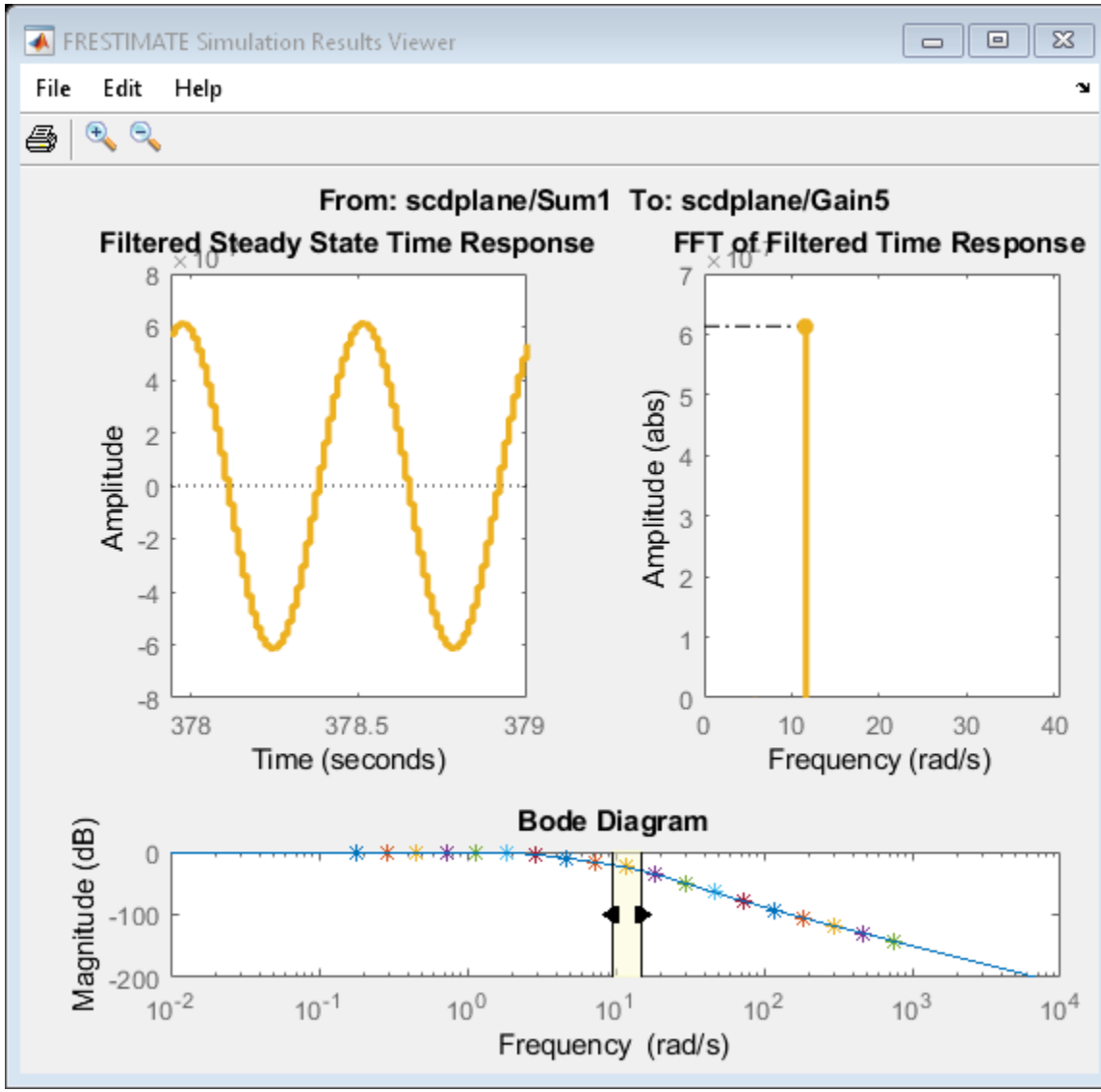
a given frequency, right-click the time response plot and make sure **Show filtered steady state output only** is selected.

Locate the source blocks in the model.

```
srcblks = frest.findSources(mdl,io);
```

Repeat the frequency response estimation with the source blocks disabled.

```
opts = frestimateOptions('BlocksToHoldConstant',srcblks);
[sysest,simout] = frestimate(mdl,io,in,opts);
frest.simView(simout,in,sysest,sys)
```


FRESTIMATE Simulation Results Viewer
File
Edit
Help
From: scdplane/Sum1 To: scdplane/Gain5
Filtered Steady State Time Response
×10
8
6
4
2
0
-2
-4
-6
-8
Amplitude
378
378.5
379
Time (seconds)
FFT of Filtered Time Response
×10
7
6
5
4
3
2
1
0
Amplitude (abs)
0
10
20
30
40
Frequency (rad/s)
Bode Diagram
Magnitude (dB)
0
-100
-200
10-2
10-1
100
101
102
103
104
Frequency (rad/s)

The resulting frequency response matches the exact linearization results.

See Also

`frest.findSources` | `frest.simView` | `frestimate` | `frestimateOptions`

More About

- “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-58

Estimate Frequency Response Models with Noise Using Signal Processing Toolbox

Open the Simulink model, and specify which portion of the model to linearize:

```
load_system('magball')
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

Create a random input signal for simulation:

```
in = frest.Random('Ts',0.001,'NumSamples',1e4);
```

Linearize the model at a steady-state operating point:

```
op = findop('magball',operspec('magball'),...
            findopOptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

Simulate the model to obtain the output at the linearization output point:

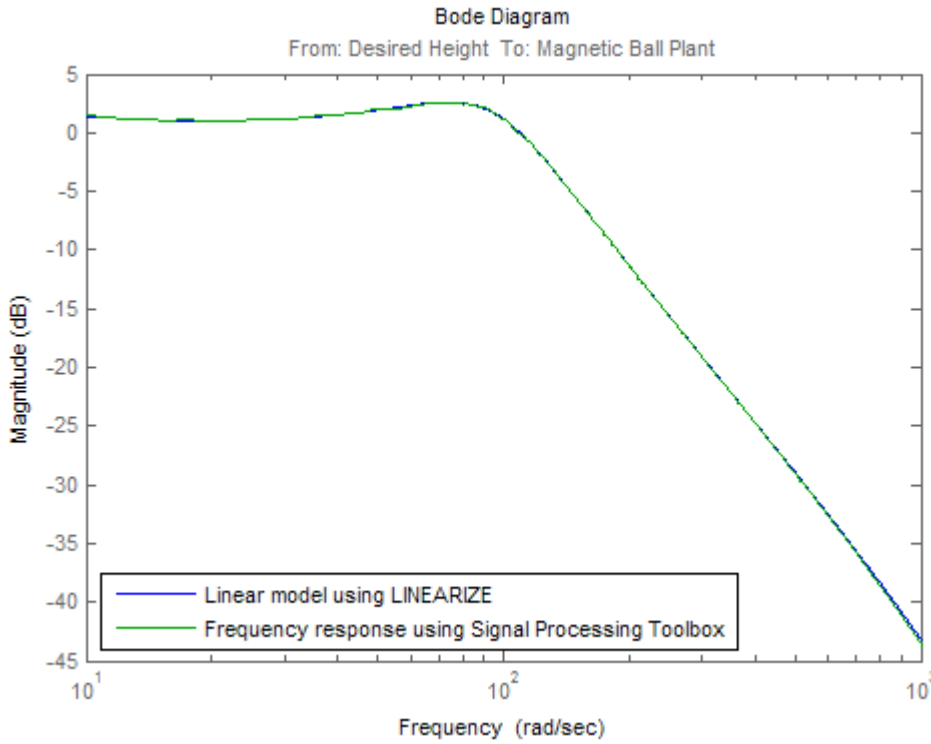
```
[sysest,simout] = frestimate('magball',io,in,op);
```

Estimate a frequency response model using Signal Processing Toolbox software, which includes windowing and averaging:

```
input = generateTimeseries(in);
output = detrend(simout{1}.Data,'constant');
[Txy,F] = tfestimate(input.Data(:),...
          output,hanning(4000),[],4000,1/in.Ts);
systfest = frd(Txy,2*pi*F);
```

Compare the results of analytical linearization and `tfestimate`:

```
ax = axes;
h = bodeplot(ax,sys,'b',systfest,'g',systfest.Frequency);
setoptions(h,'Xlim',[10,1000],'PhaseVisible','off')
legend(ax,'Linear model using LINEARIZE','Frequency response using Signal Processing Toolbox',...
          'Location','SouthWest')
```



In this case, the Signal Processing Toolbox command `tfestimate` gives a more accurate estimation than `frestimate` due to windowing and averaging.

Estimate Frequency Response Models with Noise Using System Identification Toolbox

Open the Simulink model, and specify which portion of the model to linearize:

```
load_system('magball');
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

Compute the steady-state operating point, and linearize the model:

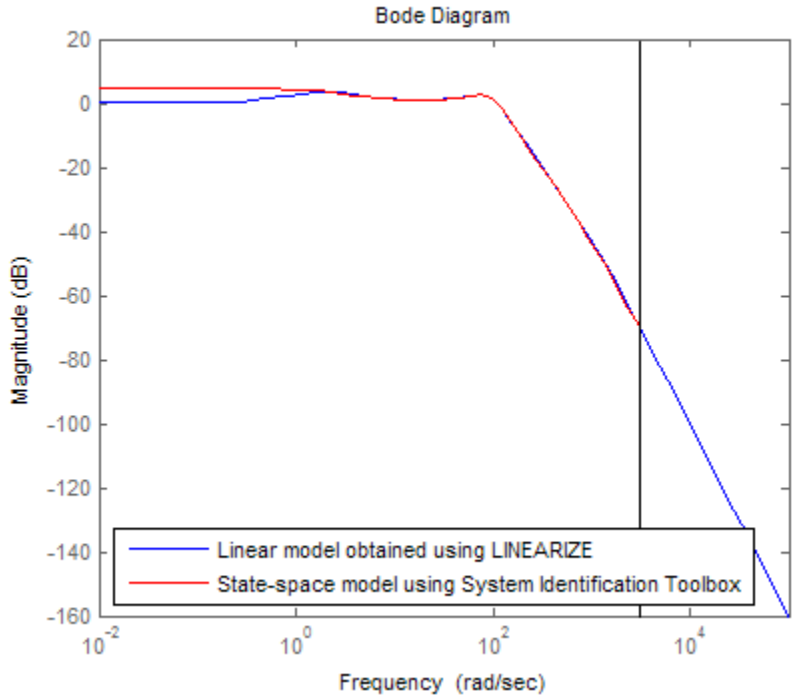
```
op = findop('magball',operspec('magball'),...
                      findopOptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

Create a chirp signal, and use it to estimate the frequency response:

```
in = frest.Chirp('FreqRange',[1 1000],...
                             'Ts',0.001,...
                             'NumSamples',1e4);
[~,simout] = frestimate('magball',io,op,in);
```

Use System Identification Toolbox software to estimate a fifth-order, state-space model. Compare the results of analytical linearization and the state-space model:

```
input = generateTimeseries(in);
output = simout{1}.Data;
data = iddata(output,input.Data(:),in.Ts);
sys_id = n4sid(detrend(data),5,'cov','none');
bodemag(sys,ss(sys_id('measured')),'r')
legend('Linear model obtained using LINEARIZE',...
       'State-space model using System Identification Toolbox',...
       'Location','SouthWest')
```

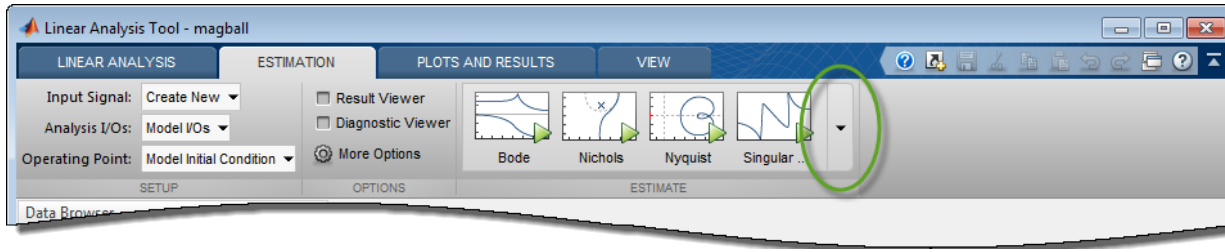

Bode Diagram
Magnitude (dB)
20
0
-20
-40
-60
-80
-100
-120
-140
-160
10^-2
10^0
10^2
10^4
Frequency (rad/sec)
Linear model obtained using LINEARIZE
State-space model using System Identification Toolbox

Generate MATLAB Code for Repeated or Batch Frequency Response Estimation

This topic shows how to generate MATLAB code for frequency response estimation from the Linear Analysis Tool. You can generate either a MATLAB script or a MATLAB function. Generated MATLAB scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB function allows you to perform multiple estimations with systematic variations in estimation parameters such as operating point (batch estimation).

To generate MATLAB code for estimation:

1. In Linear Analysis Tool, in the **Estimation** tab, interactively configure the input signal, analysis I/Os, operating point, and other parameters for frequency response estimation.
2. Click ▼ to expand the gallery.



3. Select the type of code you want to generate:

   - `Script` — Generate a MATLAB script that uses your configured parameter values. Select this option when you want to repeat the same frequency response estimation at the MATLAB command line.
   - `Function` — Generate a MATLAB function that takes analysis I/Os, operating points, and input signals as input arguments. Select this option when you want to perform multiple frequency response estimations using different parameter values (batch estimation).

To use a generated MATLAB function for batch estimation, you can create a MATLAB script with a `for` loop that cycles through values of the parameter you want to vary. Call the generated MATLAB function in each iteration of the loop.

Managing Estimation Speed and Memory

In this section...

Ways to Speed up Frequency Response Estimation

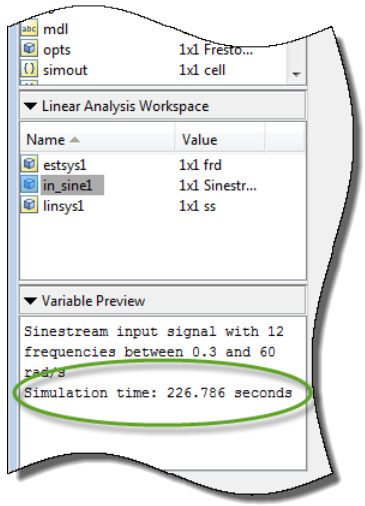
The most time consuming operation during frequency response estimation is the simulation of your Simulink model. You can try to speed up the estimation using any of the following ways:

- “Reducing Simulation Stop Time” on page 5-78
- “Specifying Accelerator Mode” on page 5-80
- “Using Parallel Computing” on page 5-80

Reducing Simulation Stop Time

The time it takes to perform frequency response estimation depends on the simulation stop time.

To obtain the simulation stop time, in the Linear Analysis tool, in the **Linear Analysis Workspace**, select the input signal. The simulation time will be displayed in the **Variable Preview**.



To obtain the simulation stop time from the input signal using MATLAB Code:

```
tfinal = getSimulationTime(input)
```

where `input` is the input signal. The simulation stop time, `tfinal`, serves as an indicator of the frequency response estimation duration.

You can reduce the simulation time by modifying your signal properties.

Input Signal	Action	Caution
Sinestream	Decrease the number of periods per frequency, `NumPeriods`, especially at lower frequencies.	You model must be at steady state to achieve accurate frequency response estimation. Reducing the number of periods might not excite your model long enough to reach steady state.
Chirp	Decrease the signal sample time, `Ts`, or the number of samples, `NumSamples`.	The frequency resolution of the estimated response depends on the number of samples `NumSamples`. Decreasing the number of samples decreases the frequency resolution of the estimated frequency response.

For information about modifying input signals, see “Modify Estimation Input Signals” on page 5-23.

Specifying Accelerator Mode

You can try to speed up frequency response estimation by specifying the Rapid Accelerator or Accelerator mode in Simulink.

For more information, see “What Is Acceleration?” (Simulink).

Using Parallel Computing

You can try to speed up frequency response estimation using parallel computing in the following situations:

- Your model has multiple inputs.
- Your single-input model uses a sinestream input signal, where the sinestream `SimulationOrder` property has the value `'OneAtATime'`.

  For information on setting this option, see the `frest.Sinestream` reference page.

In these situations, frequency response estimation performs multiple simulations. If you have installed the Parallel Computing Toolbox™ software, you can run these multiple simulations in parallel on multiple MATLAB sessions (pool of MATLAB workers).

For more information about using parallel computing, see “Speeding Up Estimation Using Parallel Computing” on page 5-80.

Speeding Up Estimation Using Parallel Computing

Configuring MATLAB for Parallel Computing

You can use parallel computing to speed up a frequency response estimation that performs multiple simulations. You can use parallel computing with the Linear Analysis Tool and `frestimate`. When you perform frequency response estimation using parallel computing, the software uses the available parallel pool. If no parallel pool is available and **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.

You can configure the software to automatically detect model dependencies and temporarily add them to the parallel pool workers. However, to ensure that workers are

able to access the undetected file and path dependencies, create a cluster profile that specifies the same. The parallel pool used to optimize the model must be associated with this cluster profile. For information on creating a cluster profile, see "Create and Modify Cluster Profiles" (Parallel Computing Toolbox).

To manually open a parallel pool that uses a specific cluster profile, use:

```
parpool(MyProfile)
```

`MyProfile` is the name of a cluster profile.

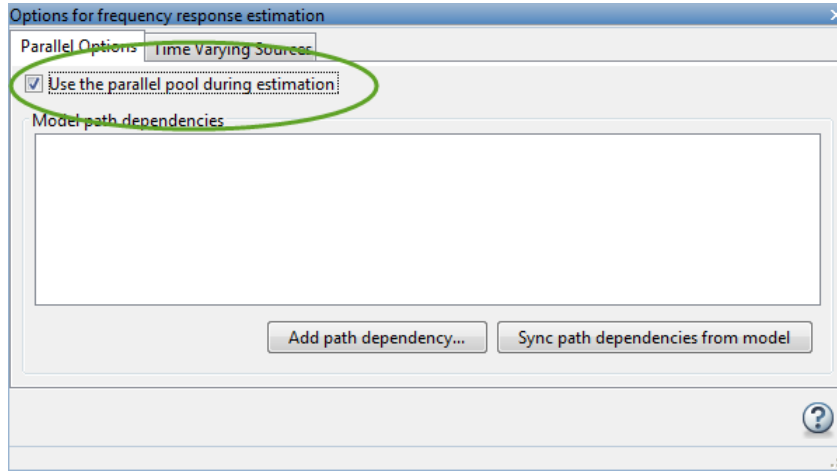
Estimating Frequency Response Using Parallel Computing Using Linear Analysis Tool

After you configure your parallel computing settings, as described in "Configuring MATLAB for Parallel Computing" on page 5-80, you can estimate the frequency response of a Simulink model using the Linear Analysis Tool.

1 In the Linear Analysis Tool, in the **Estimation** tab, click **More Options**.

This action opens the Options for frequency response estimation dialog box.

2 In the **Parallel Options** tab, select the **Use the parallel pool during estimation** check box.



3 (Optional) Click **Add path dependency**.

The Browse For Folder dialog box opens. Navigate and select the directory to add to the model path dependencies.

Click **OK**.

Tip Alternatively, manually specify the paths in the Model path dependencies list. You can specify the paths separated with a new line.

4 (Optional) Click **Sync path dependencies from model**.

This action finds the model path dependencies in your Simulink model and adds them to the **Model path dependencies** list box.

Estimating Frequency Response Using Parallel Computing (MATLAB Code)

After you configure your parallel computing settings, as described in "Configuring MATLAB for Parallel Computing" on page 5-80, you can estimate the frequency response of a Simulink model.

1 Find the paths to files that your Simulink model requires to run, called *path dependencies*.

```
dirs = frest.findDepend(model)
```

`dirs` is a cell array of character vectors containing path dependencies, such as referenced models, data files, and S-functions.

For more information about this command, see the `frest.findDepend` reference page.

To learn more about model dependencies, see "What Are Model Dependencies?" (Simulink) and "Scope of Dependency Analysis" (Simulink).

2 (Optional) Check that `dirs` includes all path dependencies. Append any missing paths to `dirs`:

```
dirs = vertcat(dirs,'\\hostname\C$\matlab\work')
```

3 (Optional) Check that all workers have access to the paths in `dirs`.

If any of the paths resides on your local drive, specify that all workers can access your local drive. For example, this command converts all references to the C drive to an equivalent network address that is accessible to all workers:

```
dirs = regexprep(dirs,'C:/','\\\\hostname\\C$\\')
```

4 Enable parallel computing and specify model path dependencies by creating an `options` object using the `frestimateOptions` command:

```
options = frestimateOptions('UseParallel','on','ParallelPathDependencies',dirs)
```

Tip To enable parallel computing for all estimations, select the global preference **Use the parallel pool when you use the "frestimate" command** check box in the MATLAB preferences. If your model has path dependencies, you must create your own frequency response options object that specifies the path dependencies before beginning estimation.

5 Estimate the frequency response:

```
[sysest,simout] = frestimate('model',io,input,options)
```

For an example of using parallel computing to speed up estimation, see “Speeding Up Frequency Response Estimation Using Parallel Computing”.

Managing Memory During Frequency Response Estimation

Frequency response estimation terminates when the simulation data exceed available memory. Insufficient memory occurs in the following situations:

- Your model performs data logging during a long simulation. A sinestream input signal with four periods at a frequency of 1e-3 rad/s runs a Simulink simulation for 25,000 s. If you are logging signals using **To Workspace** blocks, this length of simulation time might cause memory problems.
- A model with an output point discrete sample time of 1e-8 s that simulates at 5-Hz frequency (0.2 s of simulation per period), results in $\frac{0.2}{1e-8} = 2$ million samples of data per period. Typically, this amount of data requires over 300 MB of storage.

To avoid memory issues while estimating frequency response:

1 Disable any signal logging in your Simulink model.

To learn how you can identify which model components log signals and disable signal logging, see “Signal Logging” (Simulink).

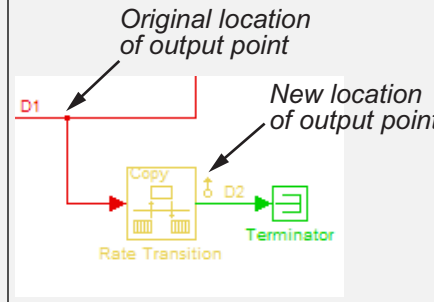
2 Try one or more of the actions listed in the following sections:

- “Model-Specific Ways to Avoid Memory Issues” on page 5-84
- “Input-Signal-Specific Ways to Avoid Memory Issues” on page 5-85

3 Repeat the estimation.

Model-Specific Ways to Avoid Memory Issues

To avoid memory issues, try one or more of the actions listed in the following table, as appropriate for your model type.

Model Type	Action
Models with fast discrete sample time specified at output point	Insert a Rate Transition block at the output point to lower the sample rate, which decreases the amount of logged data. Move the linearization output point to the output of the Rate Transition block before you estimate. Ensure that the location of the original output point does not have aliasing as a result of rate conversion.  For information on determining sample rate, see “View Sample Time Information” (Simulink). If your estimation is slow, see “Ways to Speed up Frequency Response Estimation” on page 5-78.

Model Type	Action
Models with multiple input and output points (MIMO models)	• Estimate the response for all input/output combinations separately. Then, combine the results into one MIMO model using the data format described in "Create Frequency-Response Model from Data" (Control System Toolbox). • Use parallel computing to run the independent simulations in parallel on different computers. See "Speeding Up Estimation Using Parallel Computing" on page 5-80.

Input-Signal-Specific Ways to Avoid Memory Issues

To avoid memory issues, try one or more of the actions listed in the following table, as appropriate for your input signal type.

Input Signal Type	Action
Sinestream	• Remove low frequencies from your input signal for which you do not need the frequency response. • Modify the sinestream signal to estimate each frequency separately by setting the `SimulationOrder` option to `OneAtATime`. Then estimate using a `frestimate` syntax that does not request the simulated time-response output data, for example `sysest = frestimate(model,io,input)`. • Use parallel computing to run independent simulations in parallel on different computers. See "Speeding Up Estimation Using Parallel Computing" on page 5-80. • Divide the input signal into multiple signals using `fselect`. Estimate the frequency response for each signal separately using `frestimate`. Then, combine results using `fcat`.
Chirp	Create separate input signals that divide up the swept frequency range of the original signal into smaller sections using `frest.Chirp`. Estimate the frequency response for each signal separately using `frestimate`. Then, combine results using `fcat`.
Random	Decrease the number of samples in the random input signal by changing `NumSamples` before estimating. See "Time Response Is Noisy" on page 5-55.

PID Controller Tuning

Introduction to Model-Based PID Tuning in Simulink

You can use **PID Tuner** to for interactive tuning of PID gains in a Simulink model containing a PID Controller or PID Controller (2DOF) block. **PID Tuner** allows you to achieve a good balance between performance and robustness for either one- or two-degree-of-freedom PID controllers. When you use **PID Tuner**, it:

- Automatically computes a linear model of the plant in your model. **PID Tuner** considers the plant to be the combination of all blocks between the PID controller output and input. Thus, the plant includes all blocks in the control loop, other than the controller itself. See “What Plant Does PID Tuner See?” on page 6-4.
- Automatically computes an initial PID design with a balance between performance and robustness. **PID Tuner** bases the initial design upon the open-loop frequency response of the linearized plant. See “PID Tuning Algorithm” on page 6-4.
- Provides tools and response plots to help you interactively refine the performance of the PID controller to meet your design requirements. See “Open PID Tuner” on page 6-6.

For plants that do not linearize or that linearize to zero, there are several alternatives for obtaining a plant model for tuning. These alternatives include:

- “Design PID Controller from Plant Frequency-Response Data” on page 6-49 — Use the frequency-response estimation command `frestimate` or the Frequency Response Based PID Tuner to obtain estimated frequency responses of the plant by simulation.
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73 — If you have System Identification Toolbox, you can use PID Tuner to estimate the parameters of a linear plant model based on time-domain response data. PID Tuner then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink® model.

You can use **PID Tuner** to design one- or two-degree-of-freedom PID controllers. You can often achieve both good setpoint tracking and good disturbance rejection using a one-degree-of-freedom PID controller. However, depending upon the dynamics in your model, using a one-degree-of-freedom PID controller can require a tradeoff between setpoint tracking and disturbance rejection. In such cases, if you need both good setpoint tracking and good disturbance rejection, use a two-degree-of-freedom PID Controller.

For examples of tuning one- and two-degree-of-freedom PID compensators, see:

- "PID Controller Tuning in Simulink"
- "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection" on page 6-23

What Plant Does PID Tuner See?

PID Tuner considers as the plant all blocks in the loop between the PID Controller block output and input. The blocks in your plant can include nonlinearities. Because automatic tuning requires a linear model, **PID Tuner** computes a linearized approximation of the plant in your model. This linearized model is an approximation to a nonlinear system, which is generally valid in a small region around a given operating point of the system.

By default, **PID Tuner** linearizes your plant using the initial conditions specified in your Simulink model as the operating point. The linearized plant can be of any order and can include any time delays. The **PID tuner** designs a controller for the linearized plant.

In some circumstances, however, you want to design a PID controller for a different operating point from the one defined by the model initial conditions. For example:

- The Simulink model has not yet reached steady-state at the operating point specified by the model initial conditions, and you want to design a controller for steady-state operation.
- You are designing multiple controllers for a gain-scheduling application and must design each controller for a different operating point.

In such cases, change the operating point used by **PID Tuner**. See "Opening PID Tuner" on page 6-6.

For more information about linearization, see "Linearize Nonlinear Models" on page 2-3.

PID Tuning Algorithm

Typical PID tuning objectives include:

- Closed-loop stability — The closed-loop system output remains bounded for bounded input.
- Adequate performance — The closed-loop system tracks reference changes and suppresses disturbances as rapidly as possible. The larger the loop bandwidth (the frequency of unity open-loop gain), the faster the controller responds to changes in the reference or disturbances in the loop.

- Adequate robustness — The loop design has enough gain margin and phase margin to allow for modeling errors or variations in system dynamics.

MathWorks algorithm for tuning PID controllers meets these objectives by tuning the PID gains to achieve a good balance between performance and robustness. By default, the algorithm chooses a crossover frequency (loop bandwidth) based on the plant dynamics, and designs for a target phase margin of 60°. When you interactively change the response time, bandwidth, transient response, or phase margin using the **PID Tuner** interface, the algorithm computes new PID gains.

For a given robustness (minimum phase margin), the tuning algorithm chooses a controller design that balances the two measures of performance, reference tracking and disturbance rejection. You can change the design focus to favor one of these performance measures. To do so, use the **Options** dialog box in **PID Tuner**.

When you change the design focus, the algorithm attempts to adjust the gains to favor either reference tracking or disturbance rejection, while achieving the same minimum phase margin. The more tunable parameters there are in the system, the more likely it is that the PID algorithm can achieve the desired design focus without sacrificing robustness. For example, setting the design focus is more likely to be effective for PID controllers than for P or PI controllers. In all cases, fine-tuning the performance of the system depends strongly on the properties of your plant. For some plants, changing the design focus has little or no effect.

Open PID Tuner

Prerequisites for PID Tuning

Before you can use **PID Tuner**, you must:

- Create a Simulink model containing a PID Controller or PID Controller (2DOF) block. Your model can have one or more PID blocks, but you can only tune one PID block at a time.

  - If you are tuning a multi-loop control system with coupling between the loops, consider using other Simulink Control Design tools instead of **PID Tuner**. See **Control System Designer** and "Cascaded Multi-Loop/Multi-Compensator Feedback Design" on page 8-62 for more information.
  - The PID Controller blocks support vector signals. However, using **PID Tuner** requires scalar signals at the block inputs. That is, the PID block must represent a single PID controller.

  Your plant (all blocks in the control loop other than the controller) can be linear or nonlinear. The plant can also be of any order, and have any time delays.
- Configure the PID block settings, such as controller type, controller form, time domain, sample time. See the PID Controller or PID Controller (2DOF) block reference pages for more information about configuring these settings.

Opening PID Tuner

To open **PID Tuner** and view the initial compensator design:

1. Open the Simulink model by typing the model name at the MATLAB command prompt.
2. Double-click the PID Controller block to open the block dialog box.
3. In the block dialog box, in the **Select Tuning Method** drop-down list, select `Transfer Function Based (PID Tuner App)`. Click **Tune** to open **PID Tuner**.

When you open **PID Tuner**, the following actions occur:

- **PID Tuner** automatically linearizes the plant at the operating point specified by the model initial conditions, as described in "What Plant Does PID Tuner See?" on page 6-

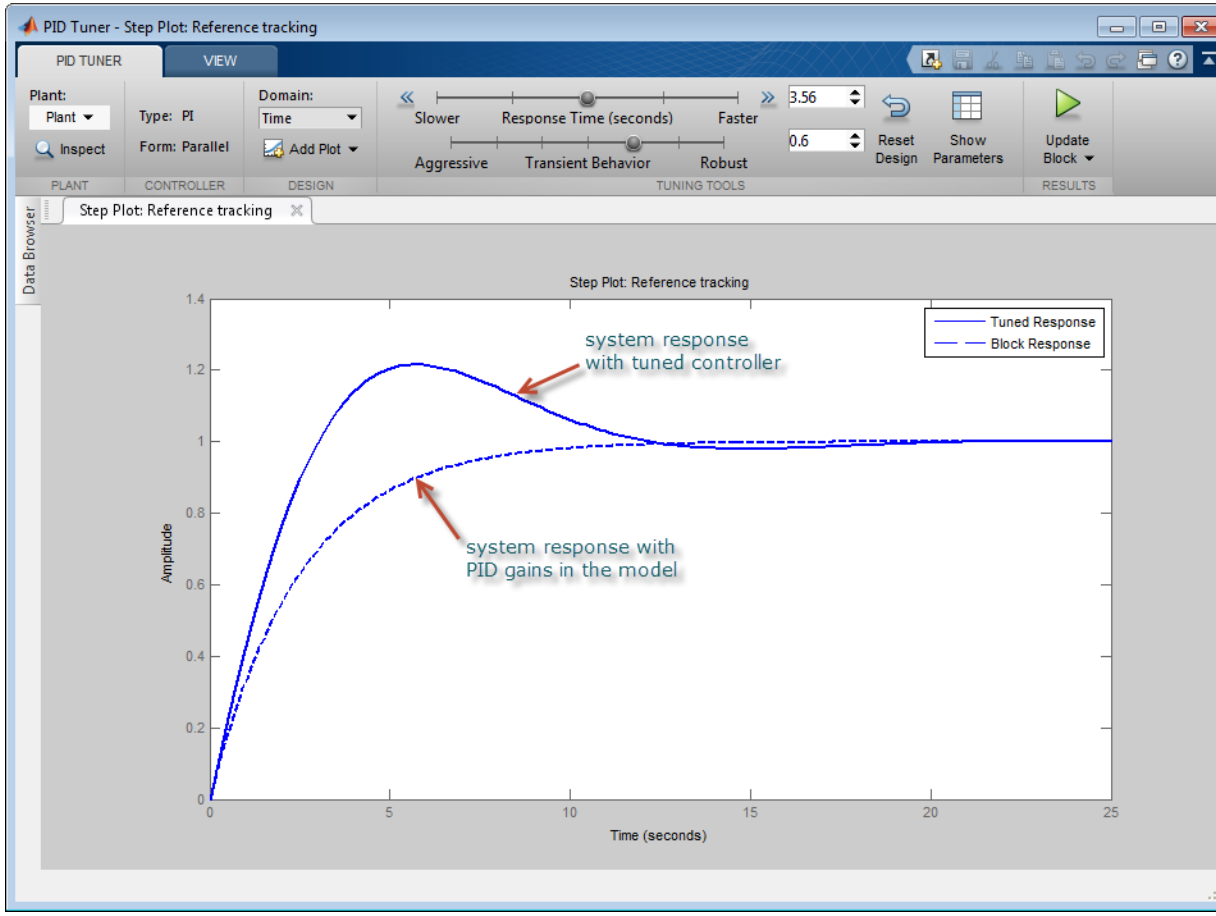
4. If you want to design a controller for a different operating point, see “Tune at a Different Operating Point” on page 6-19.

Note If the plant model in the PID loop linearizes to zero, **PID Tuner** provides the **Obtain plant model** dialog box. This dialog box allows you to obtain a new plant model by either:

- Linearizing at a different operating point (see “Tune at a Different Operating Point” on page 6-19).
- Importing an LTI model object representing the plant. For example, you can import frequency response data (an `frd` model) obtained by frequency response estimation. For more information, see “Design PID Controller Using FRD Model Obtained From "frestimate" Command” on page 6-102.
- Identifying a linear plant model from simulated or measured response data (requires System Identification Toolbox software). **PID Tuner** uses system identification to estimate a linear plant model from the time-domain response of your plant to an applied input. For an example, see “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73.

As an alternative, you can exit **PID Tuner** and use the **Frequency Response Based PID Tuner**, which runs simulations to perturb the plant and estimate frequency responses at frequencies near the control bandwidth. See “Frequency Response Based Tuning Basics” on page 6-51.

- **PID Tuner** computes an initial compensator design for the linearized plant model using the algorithm described in “PID Tuning Algorithm” on page 6-4.
- **PID Tuner** displays the closed-loop step reference tracking response for the initial compensator design. For comparison, the display also includes the closed-loop response for the gains specified in the PID Controller block, if that closed loop is stable, as shown in the following figure.



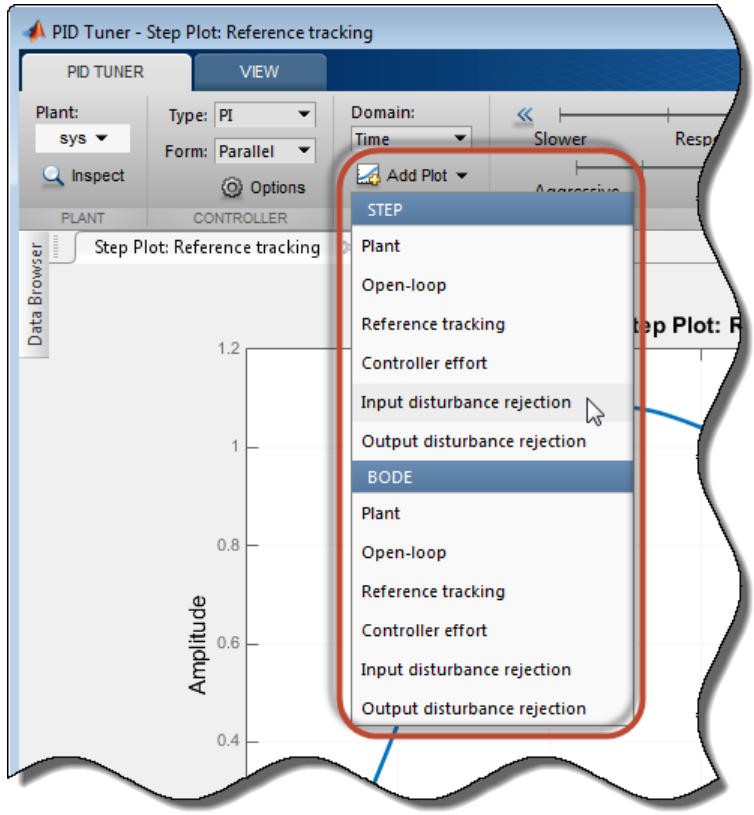
Tip After the tuner opens, you can close the PID Controller block dialog box.

Analyze Design in PID Tuner

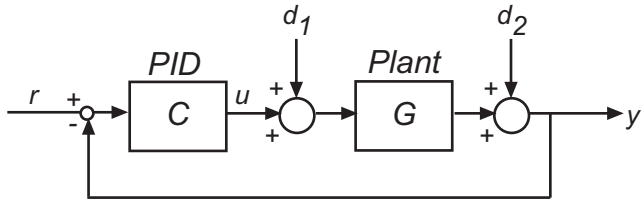
In this section...

Plot System Responses

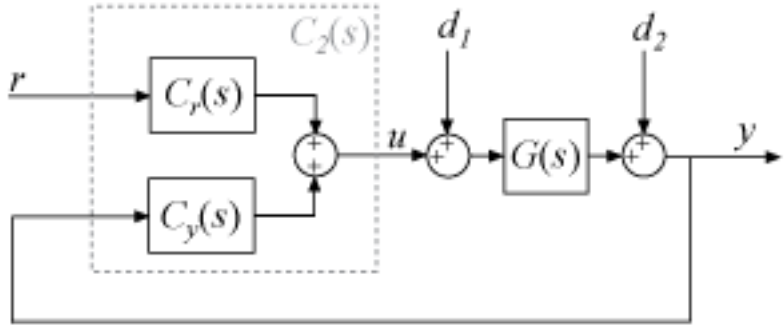
To determine whether the compensator design meets your requirements, you can analyze the system response using the response plots. On the **PID Tuner** tab, select a response plot from the **Add Plot** menu. The **Add Plot** menu also lets you choose from several step plots (time-domain response) or Bode plots (frequency-domain response).



For 1-DOF PID controller types such as PI, PIDF, and PDF, **PID Tuner** computes system responses based upon the following single-loop control architecture:



For 2-DOF PID controller types such as PI2, PIDF2, and I-PD, **PID Tuner** computes responses based upon the following architecture:



The system responses are based on the decomposition of the 2-DOF PID controller, C_2, into a setpoint component *Cr* and a feedback component *Cy*, as described in "Two-Degree-of-Freedom PID Controllers" (Control System Toolbox).

The following table summarizes the available responses for analysis plots in **PID Tuner**.

Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
`Plant`	G	G	Shows the plant response. Use to examine plant dynamics.
`Open-loop`	GC	$-GC_y$	Shows response of the open-loop controller-plant system. Use for frequency-domain design. Use when your design specifications include robustness criteria such as open-loop gain margin and phase margin.

Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
`Reference tracking`	$\frac{GC}{1+GC}$ (from r to y)	$\frac{GC_r}{1-GC_y}$ (from r to y)	Shows the closed-loop system response to a step change in setpoint. Use when your design specifications include setpoint tracking.
`Controller effort`	$\frac{C}{1+GC}$ (from r to u)	$\frac{C_r}{1-GC_y}$ (from r to u)	Shows the closed-loop controller output response to a step change in setpoint. Use when your design is limited by practical constraints, such as controller saturation.
`Input disturbance rejection`	$\frac{G}{1+GC}$ (from d_1 to y)	$\frac{G}{1-GC_y}$ (from d_1 to y)	Shows the closed-loop system response to load disturbance (a step disturbance at the plant input). Use when your design specifications include input disturbance rejection.
`Output disturbance rejection`	$\frac{1}{1+GC}$ (from d_2 to y)	$\frac{1}{1-GC_y}$ (from d_2 to y)	Shows the closed-loop system response to a step disturbance at plant output. Use when you want to analyze sensitivity to measurement noise.

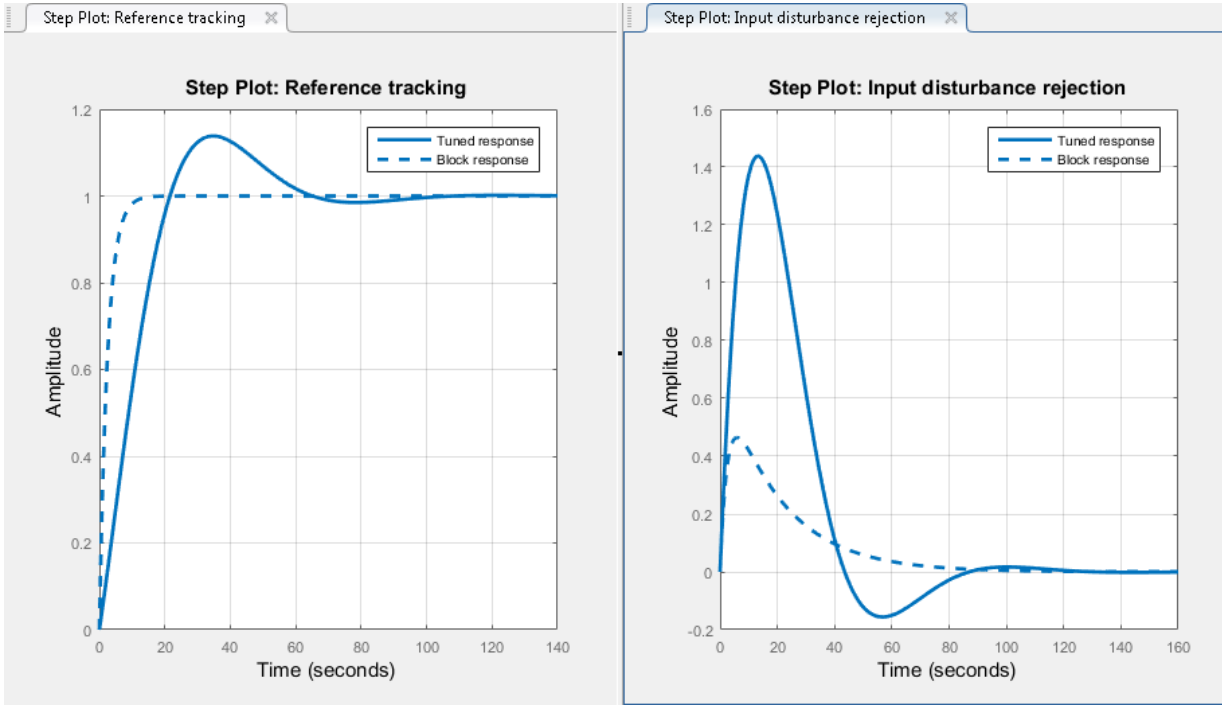
Compare Tuned Response to Block Response

By default, **PID Tuner** plots system responses using both:

- The PID coefficient values in the PID Controller block in the Simulink model (Block response).

- The PID coefficient values of the current **PID Tuner** design (Tuned response).

As you adjust the current **PID Tuner** design, such as by moving the sliders, the Tuned response plots change, while the Block response plots do not.



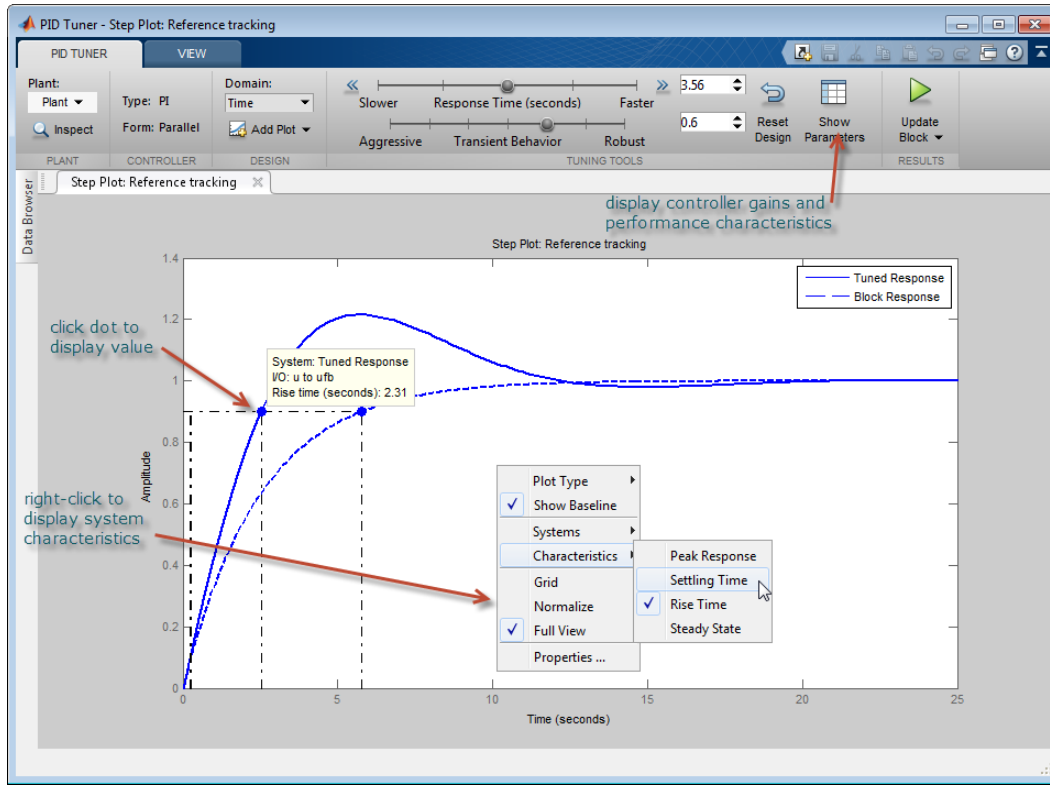
To write the current **PID Tuner** design to the Simulink model, click  . When you do so, the current Tuned response becomes the Block response. Further adjustment of the current design creates a new Tuned response line.

To hide the Block response, click  **Options**, and uncheck **Show Block Response**.

View Numeric Values of System Characteristics

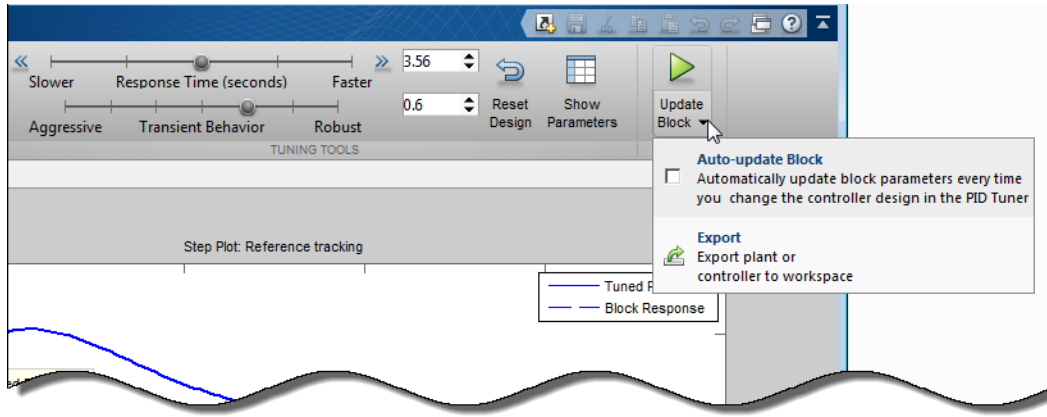
You can view the values for system characteristics, such as peak response and gain margin, either:

- Directly on the response plot — Use the right-click menu to add characteristics, which appear as blue markers. Then, left-click the marker to display the corresponding data panel.
- In the **Performance and robustness** table — To display this table, click **Show Parameters**.



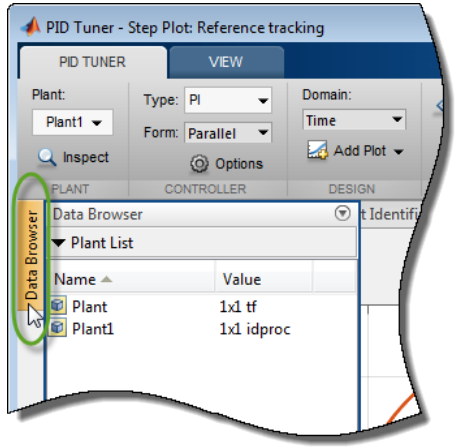
Export Plant or Controller to MATLAB Workspace

You can export the linearized plant model computed by **PID Tuner** to the MATLAB workspace for further analysis. To do so, click **Update Block** and select **Export**.

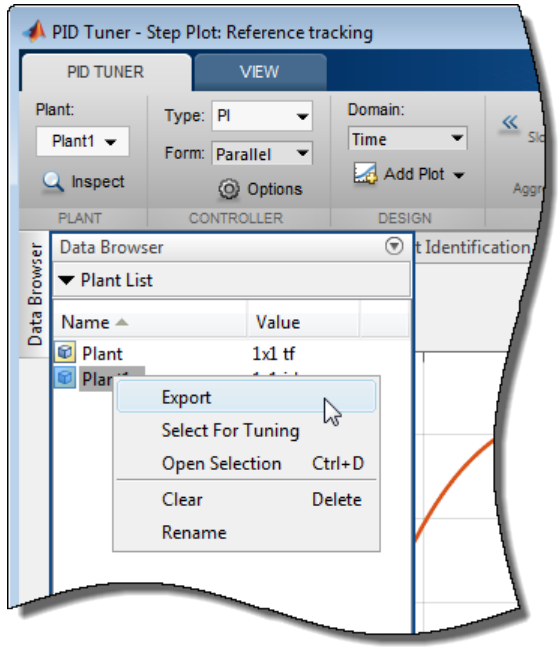


In the Export dialog box, check the models that you want to export. Click **OK** to export the plant or controller to the MATLAB workspace as state-space (`ss`) model object or `pid` object, respectively.

Alternatively, you can export a model using the context menu in the **Data Browser**. To do so, click the **Data Browser** tab.



Then, right-click the model and select **Export**.



Refine the Design

If the response of the initial controller design does not meet your requirements, you can interactively adjust the design. **PID Tuner** gives you two **Domain** options for refining the controller design:

- `Time` domain (default) — Use the **Response Time** slider to make the closed-loop response of the control system faster or slower. Use the **Transient Behavior** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.
- `Frequency` — Use the **Bandwidth** slider to make the closed-loop response of the control system faster or slower (the response time is $2/w_c$, where w_c is the bandwidth). Use the **Phase Margin** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

In both modes, there is a tradeoff between reference tracking and disturbance rejection performance. For an example that shows how to use the sliders to adjust this tradeoff,

see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” on page 6-23.

Once you find a compensator design that meets your requirements, verify that it behaves in a similar way in the nonlinear Simulink model. For instructions, see “Verify the PID Design in Your Simulink Model” on page 6-18.

Tip To revert to the initial controller design after moving the sliders, click **Reset Design**.

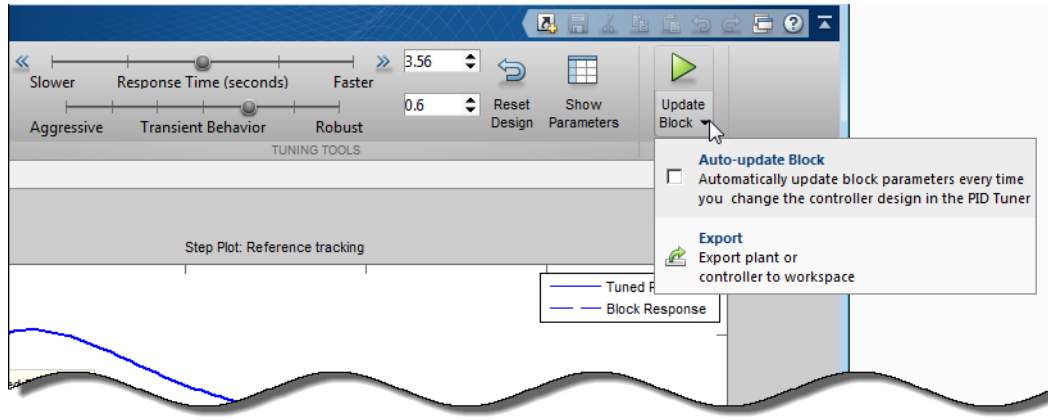
Verify the PID Design in Your Simulink Model

In **PID Tuner**, you tune the compensator using a linear model of your plant. First, you find a good compensator design in **PID Tuner**. Then, verify that the tuned controller meets your design requirements when applied to the nonlinear plant in your Simulink model.

To verify the compensator design in the nonlinear Simulink model:

1. In the **PID Tuner** tab, click  to update the Simulink PID Controller block with the tuned PID parameters.

   Tip To update PID block parameters automatically as you tune the controller in **PID Tuner**, click **Update Block** and check **Auto-update block**.

2. Simulate the Simulink model, and evaluate whether the simulation output meets your design requirements.

Because **PID Tuner** works with a linear model of your plant, the simulated response sometimes does not match the response in **PID Tuner**. See “Simulated Response Does Not Match the PID Tuner Response” on page 6-131 for more information.

If the simulated response does not meet your design requirements, see “Cannot Find an Acceptable PID Design in the Simulated Model” on page 6-133.

Tune at a Different Operating Point

By default, **PID Tuner** linearizes your plant and designs a controller at the operating point specified by the initial conditions in your Simulink model. Sometimes, this operating point differs from the operating point for which you want to design a controller. For example, you want to design a controller for your system at steady-state. However, the Simulink model is not generally at steady-state at the initial condition. In this case, change the operating point that **PID Tuner** uses for linearizing your plant and designing a controller.

To set a new operating point for **PID Tuner**, use one of the following methods. The method you choose depends upon the information you have about your desired operating point.

In this section...
"Known State Values Yield the Desired Operating Conditions" on page 6-19
"Model Reaches Desired Operating Conditions at a Finite Time" on page 6-19
"You Computed an Operating Point in the Linear Analysis Tool" on page 6-20

Known State Values Yield the Desired Operating Conditions

In this case, set the state values in the model directly.

1. Close **PID Tuner**.
2. Set the initial conditions of the components of your model to the values that yield the desired operating conditions.
3. Click **Tune** in the PID Controller dialog box to open **PID Tuner**. **PID Tuner** linearizes the plant using the new default operating point and designs a new initial controller for the new linear plant model.

After **PID Tuner** generates a new initial controller design, continue from "Analyze Design in PID Tuner" on page 6-9.

Model Reaches Desired Operating Conditions at a Finite Time

In this case, use **PID Tuner** to relinearize the model at a particular simulation time.

1 In the **PID Tuner** tab, in the **Plant** menu, select **Re-linearize Closed Loop**.

2 In the **Closed Loop Re-Linearization** tab, click **Run Simulation** to simulate the model for the time specified in the **Simulation Time** text box.

PID Tuner plots the error signal as a function of time. You can use this plot to identify a time at which the model is in steady-state. Slide the vertical bar to a snapshot time at which you want to linearize the model.

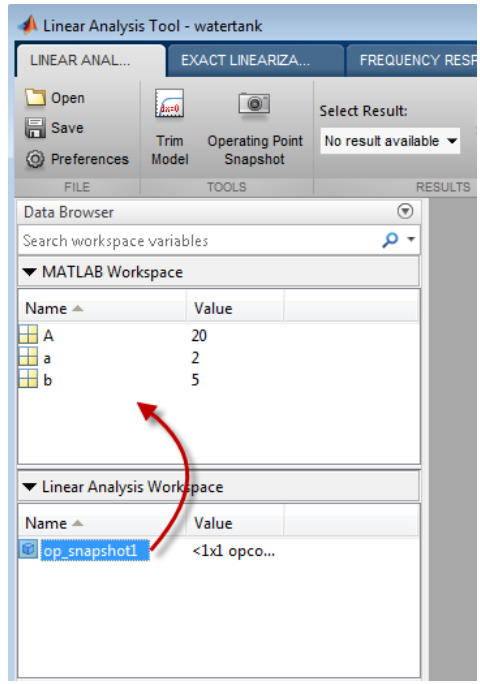
3 Click **Linearize** to linearize the model at the selected snapshot time. **PID Tuner** computes a new linearized plant and saves it to the **PID Tuner** workspace. **PID Tuner** automatically designs a controller for the new plant, and displays a response plot for the new closed-loop system. **PID Tuner** returns you **PID Tuner** tab, where the **Plant** menu reflects that the new plant is selected for the current controller design.

Note For models with Trigger-Based Operating Point Snapshot blocks, the software captures an operating point at events that trigger before the simulation reaches the snapshot time.

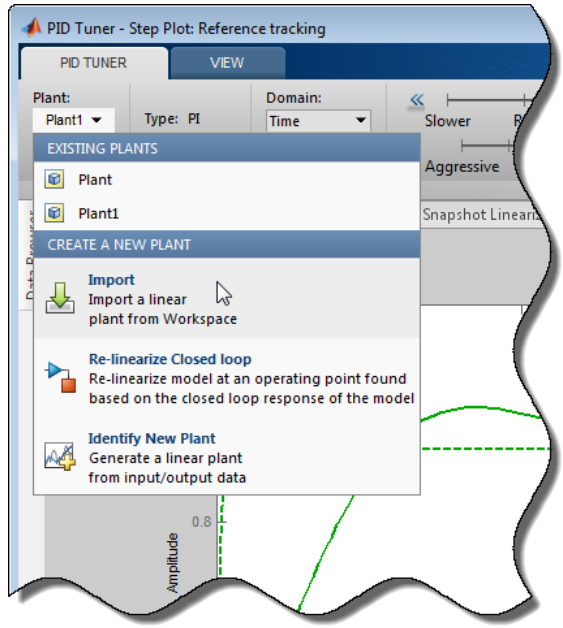
After **PID Tuner** generates a new initial controller design, continue from “Analyze Design in PID Tuner” on page 6-9.

You Computed an Operating Point in the Linear Analysis Tool

1 In the Linear Analysis tool, drag the saved operating point object from the Linear Analysis Workspace to the MATLAB Workspace.



2 In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select **Import**.



3 Select **Importing an LTI system or linearizing at an operating point defined in MATLAB workspace**. Select your exported operating point in the table.

4 Click **OK**. **PID Tuner** computes a new linearized plant and saves it to the **PID Tuner** workspace. **PID Tuner** automatically designs a controller for the new plant, and displays a response plot for the new closed-loop system. **PID Tuner** returns you **PID Tuner** tab, where the **Plant** menu reflects that the new plant is selected for the current controller design.

After **PID Tuner** generates a new initial controller design, continue from “Analyze Design in PID Tuner” on page 6-9.

See Also

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Points” on page 1-6

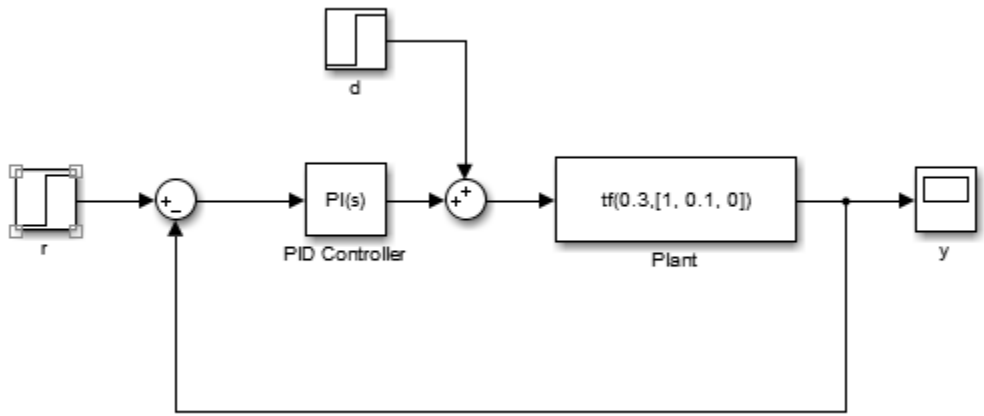
Tune PID Controller to Favor Reference Tracking or Disturbance Rejection

This example shows how to tune a PID controller to reduce overshoot in reference tracking or to improve rejection of a disturbance at the plant input. Using the **PID Tuner** app, the example illustrates the tradeoff between reference tracking and disturbance-rejection performance in PI and PID control systems.

Design Initial PI Controller

Load a Simulink model that contains a PID Controller block.

```
open_system('singlePIloop')
```

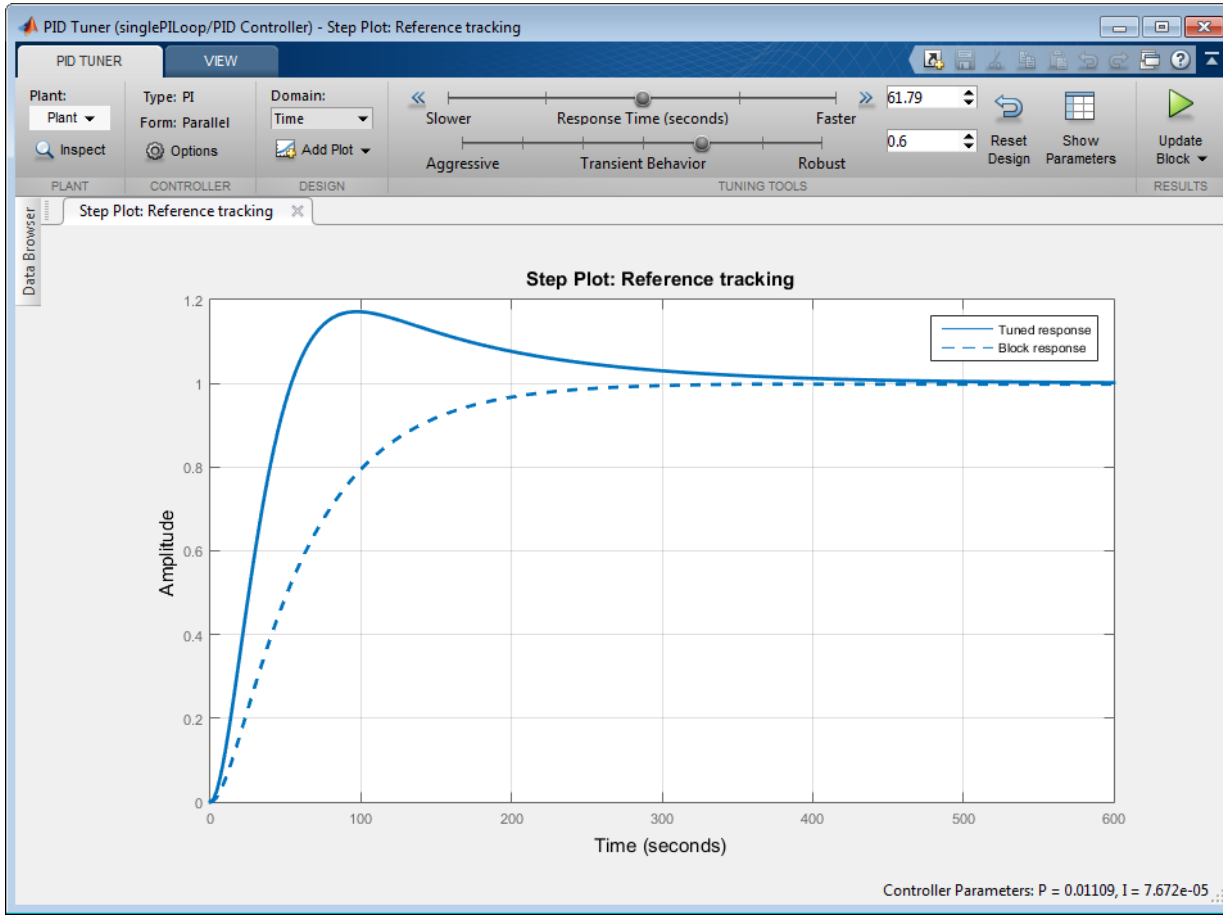


The plant in this example is:

$$\text{Plant} = \frac{0.3}{s^2 + 0.1s}.$$

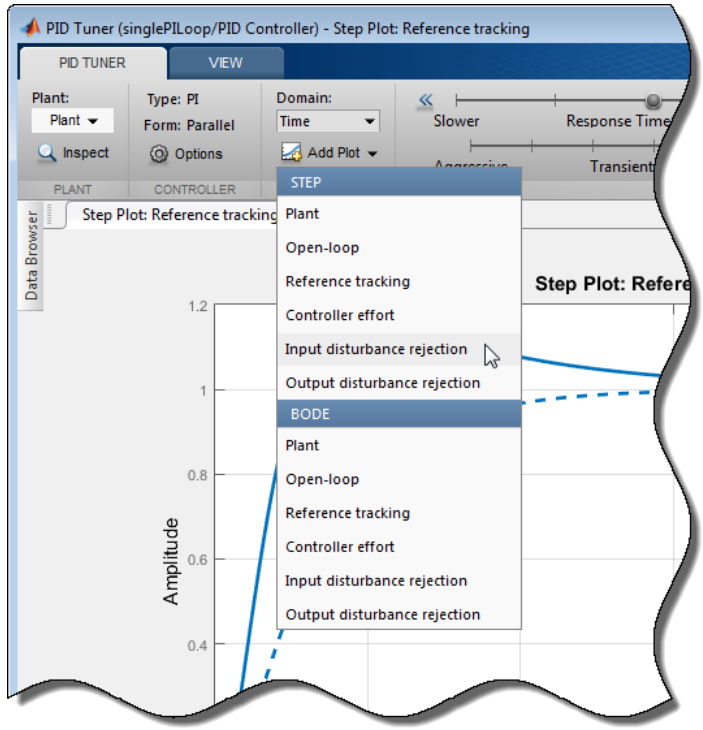
The model also includes a reference signal and a step disturbance at the plant input. Reference tracking is the response at `y` to the reference signal, `r`. Disturbance rejection is a measure of the suppression at `y` of the injected disturbance, `d`. When you use **PID Tuner** to tune the controller, you can adjust the design to favor reference tracking or disturbance rejection as your application requires.

Design an initial controller for the plant. To do so, double-click the PID Controller block to open the Block Parameters dialog box, and click **Tune**. **PID Tuner** opens and automatically computes an initial controller design.

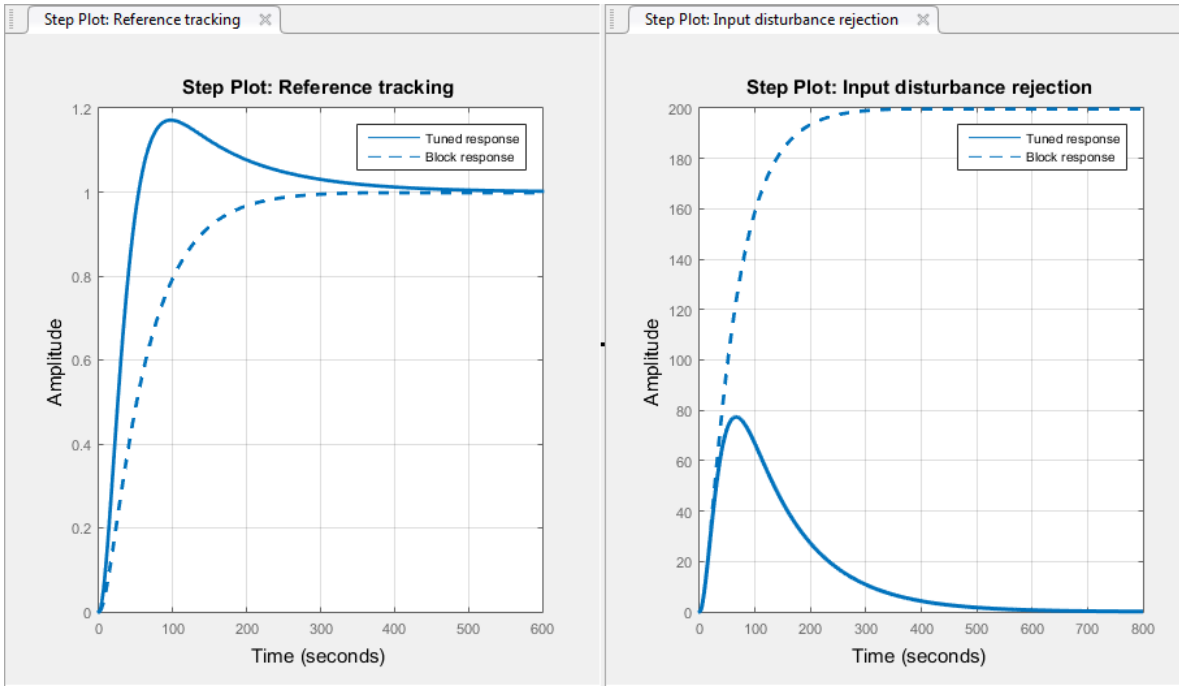


The PID Controller in the Simulink model is configured as a PI-type controller. Therefore, the initial controller designed by **PID Tuner** is also of PI-type.

Add a step response plot of the input disturbance rejection. Select **Add Plot** > **Input Disturbance Rejection**.



PID Tuner tiles the disturbance-rejection plot side by side with the reference-tracking plot.



Tip Use the options in the **View** tab to change how **PID Tuner** displays multiple plots.

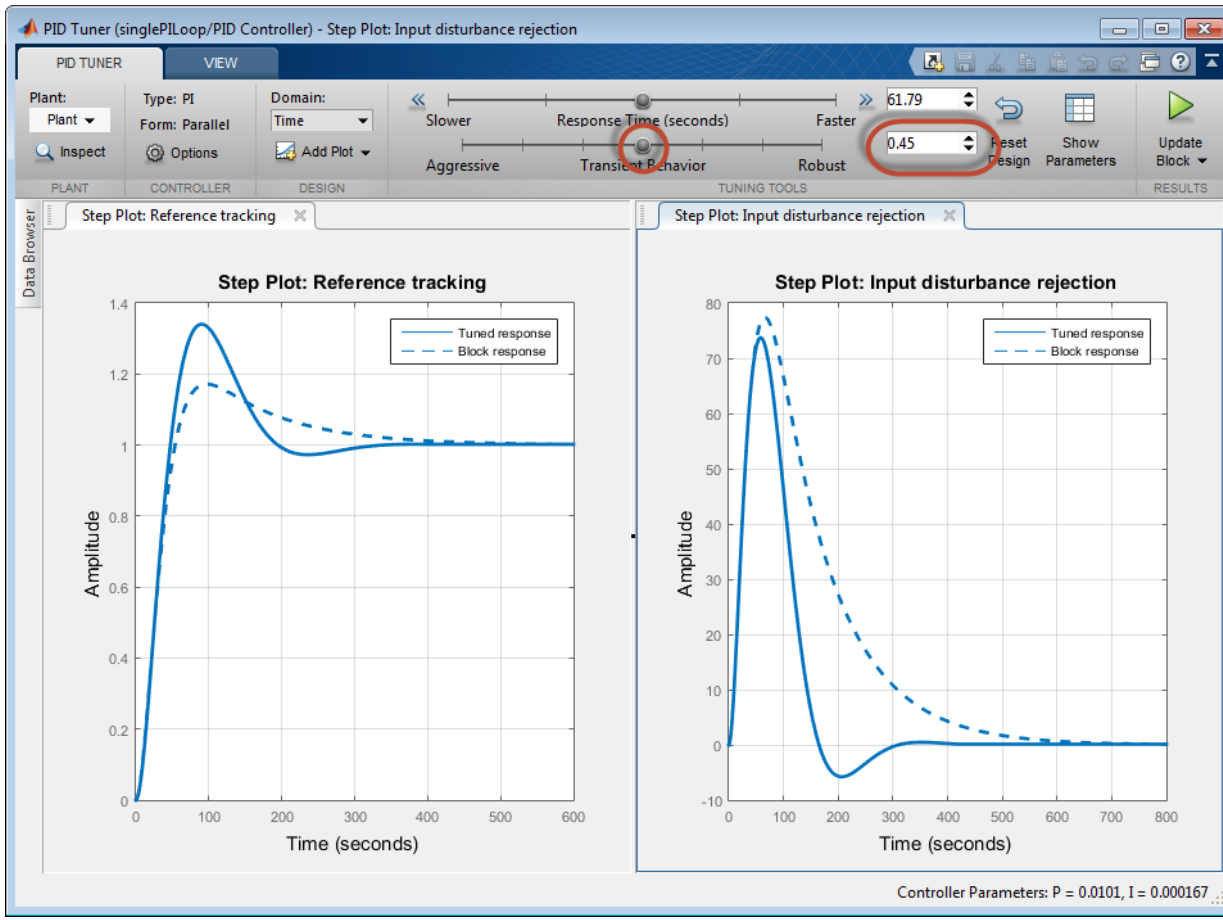
By default, for a given bandwidth and phase margin, **PID Tuner** tunes the controller to achieve a balance between reference tracking and disturbance rejection. In this case, the controller yields some overshoot in the reference-tracking response. The controller also suppresses the input disturbance with a longer settling time than the reference tracking, after an initial peak.

Click  to update the Simulink model with this initial controller design. Doing so also updates the Block Response plots in **PID Tuner**, so that as you change the controller design, you can compare the results with the initial design.

Adjust Transient Behavior

Depending on your application, you might want to alter the balance between reference tracking and disturbance rejection to favor one or the other. For a PI controller, you can

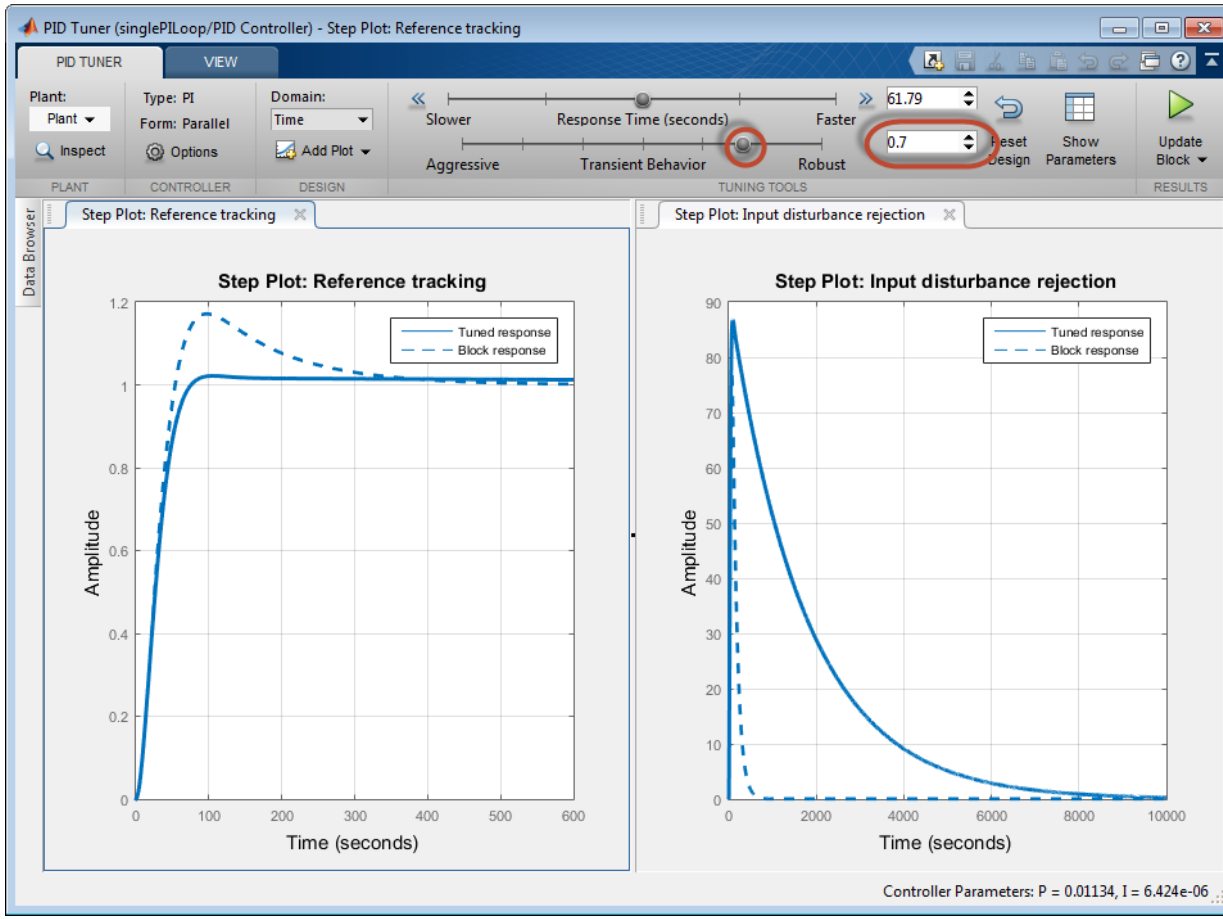
alter this balance using the **Transient Behavior** slider. Move the **Transient behavior** slider to the left to improve the disturbance rejection. The responses with the initial controller design are now displayed as the `Block response` (dotted line).



Lowering the transient-behavior coefficient to 0.45 speeds up disturbance rejection, but also increases overshoot in the reference-tracking response.

Tip Right-click the reference-tracking plot and select **Characteristics > Peak Response** to obtain a numerical value for the overshoot.

Move the **Transient behavior** to the right until the overshoot in the reference-tracking response is minimized.

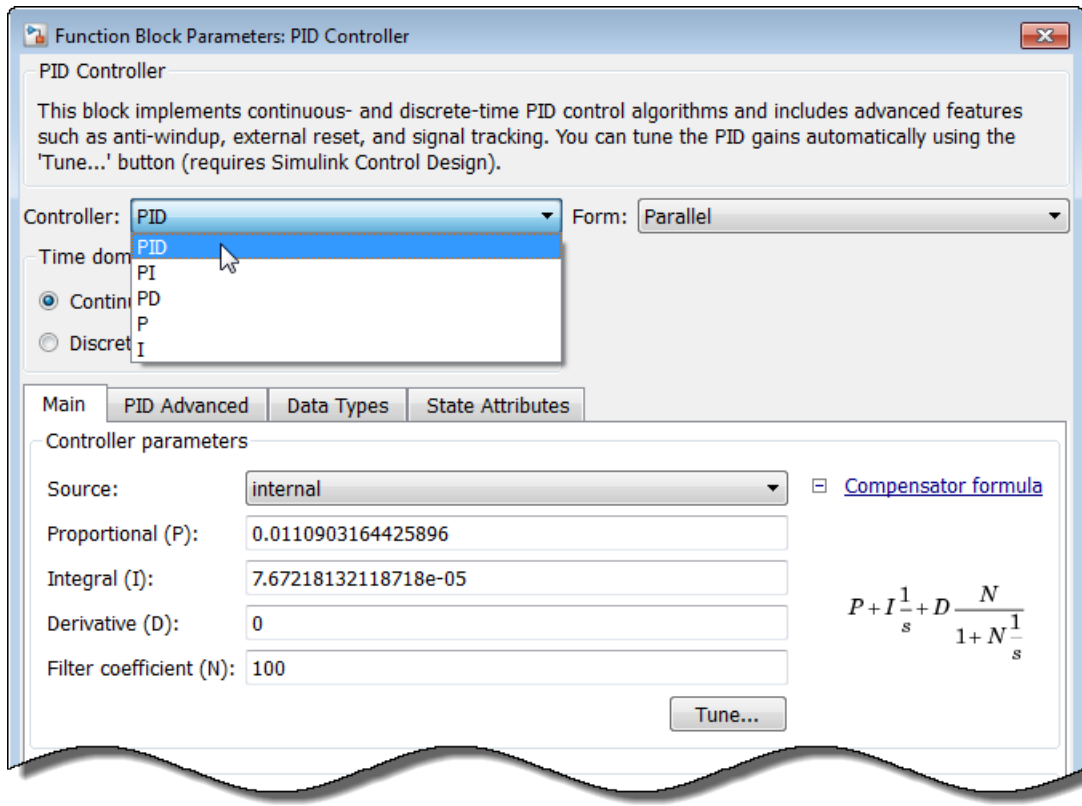


Increasing the transient-behavior coefficient to 0.70 nearly eliminates the overshoot, but results in sluggish disturbance rejection. You can try moving the **Transient behavior** slider until you find a suitable balance between reference tracking and disturbance rejection for your application. How much the slider affects the balance depends on the plant model. For some plant models, the effect is not as large as shown in this example.

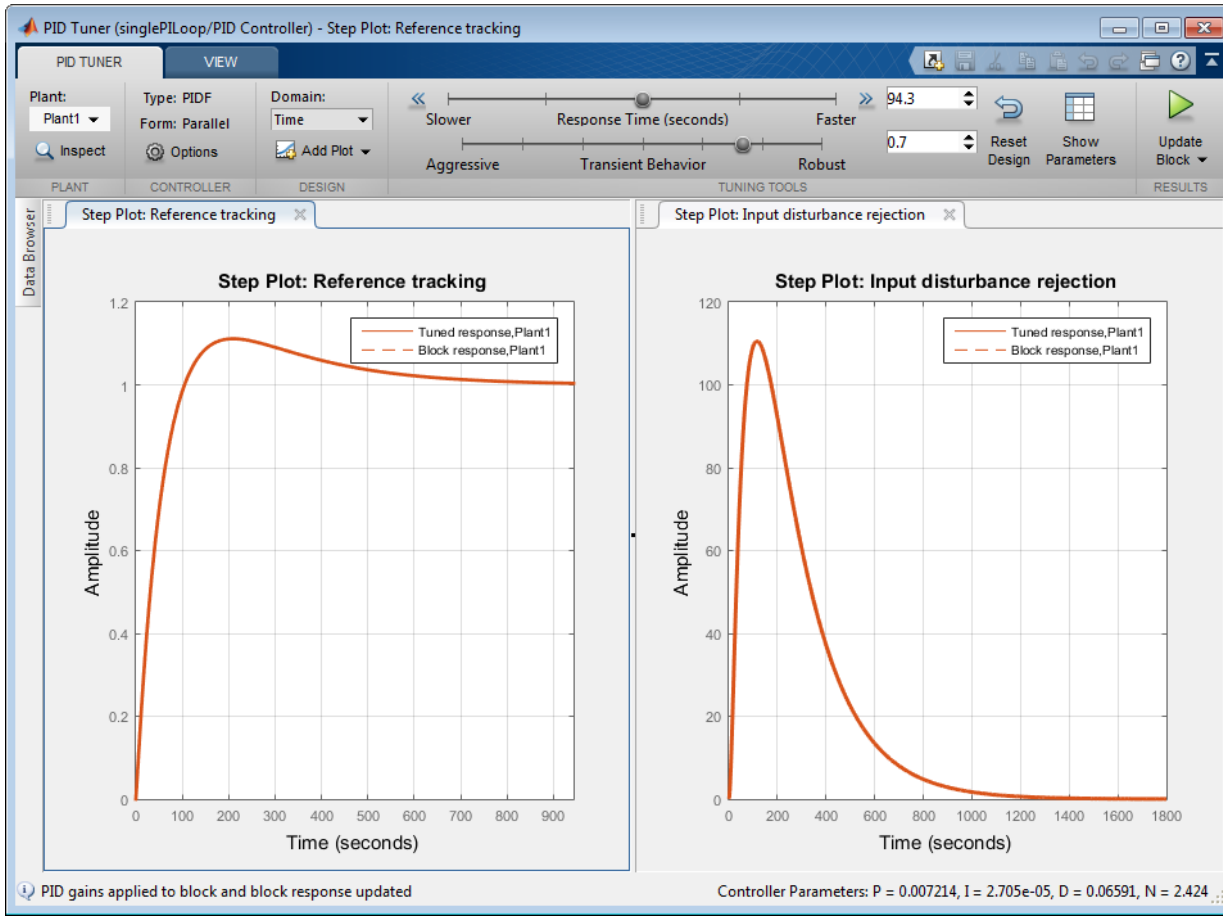
Change PID Tuning Design Focus

So far, the response time of the control system has remained fixed while you have changed the transient-behavior coefficient. These operations are equivalent to fixing the bandwidth and varying the target minimum phase margin of the system. If you want to fix both the bandwidth and target phase margin, you can still change the balance between reference tracking and disturbance rejection. To tune a controller that favors either disturbance rejection or reference tracking, you change the design focus of the PID tuning algorithm.

Changing the **PID Tuner** design focus is more effective the more tunable parameters there are in the control system. Therefore, it does not have much effect when used with a PI controller. To see its effect, change the controller type to PID. In the Simulink model, double-click the PID controller block. In the block parameters dialog box, in the **Controller** drop-down menu, select `PID`.

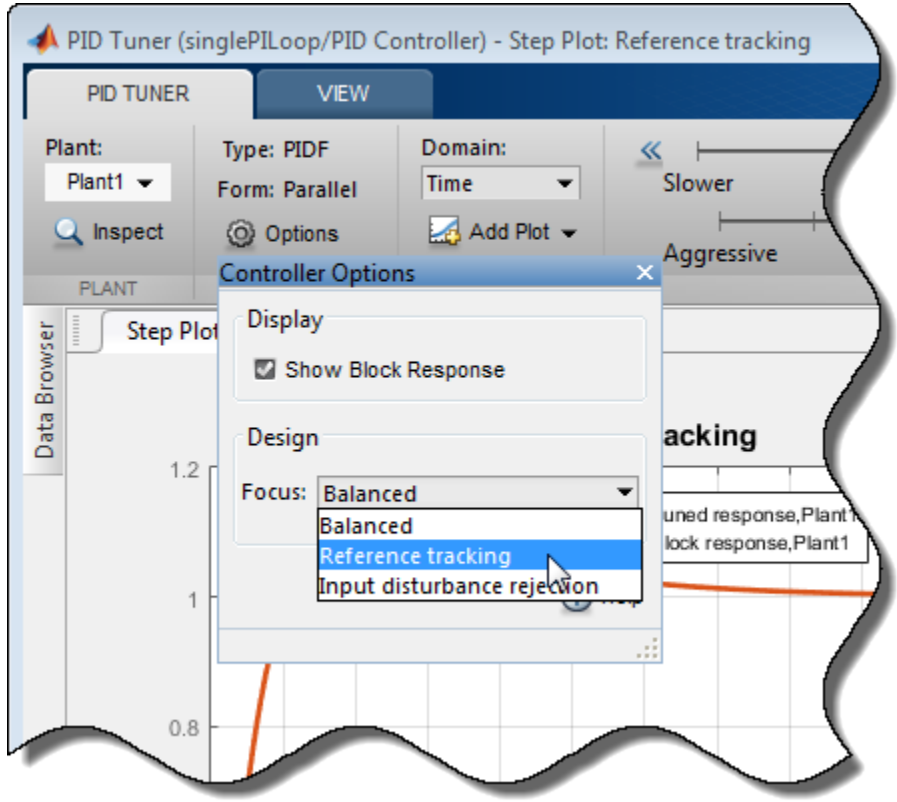


Click **Apply**. Then, click **Tune**. This action updates **PID Tuner** with a new controller design, this time for a PID controller. Click  to the Simulink model with this initial PID controller design, so that you can compare the results when you change design focus.

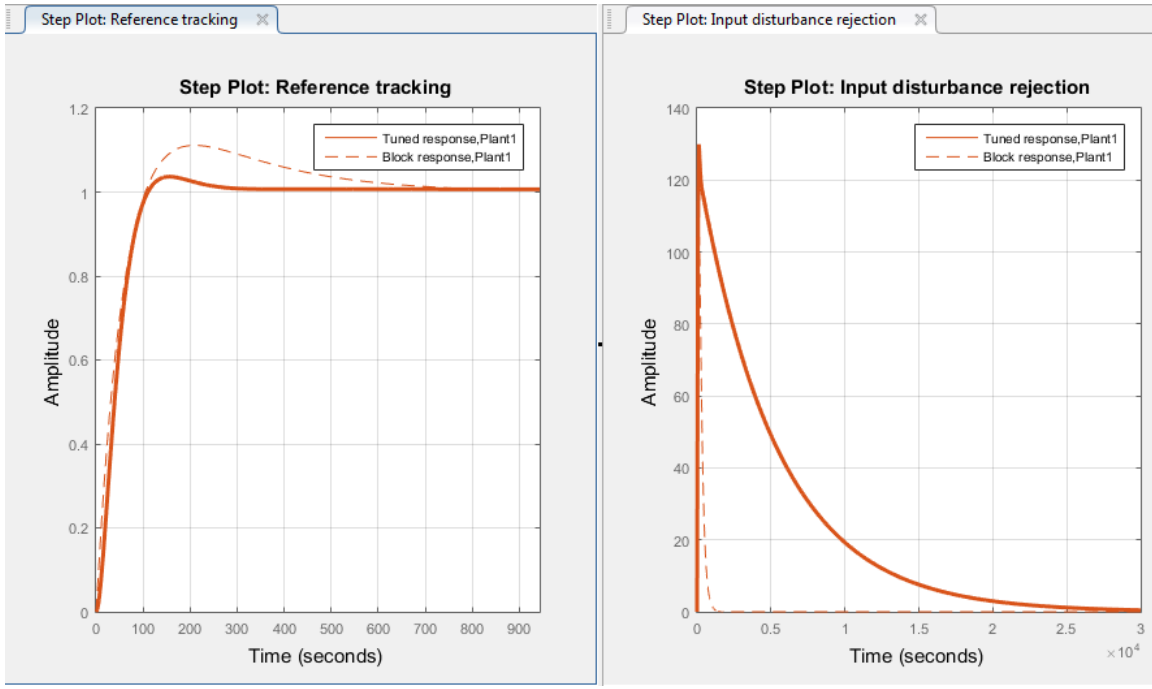


As in the PI case, the initial PID design balances reference tracking and disturbance rejection. In this case as well, the controller yields some overshoot in the reference-tracking response, and suppresses the input disturbance with a longer settling time.

Change the **PID Tuner** design focus to favor reference tracking without changing the response time or the transient-behavior coefficient. To do so, click **Options**, and in the **Focus** menu, select `Reference tracking`.

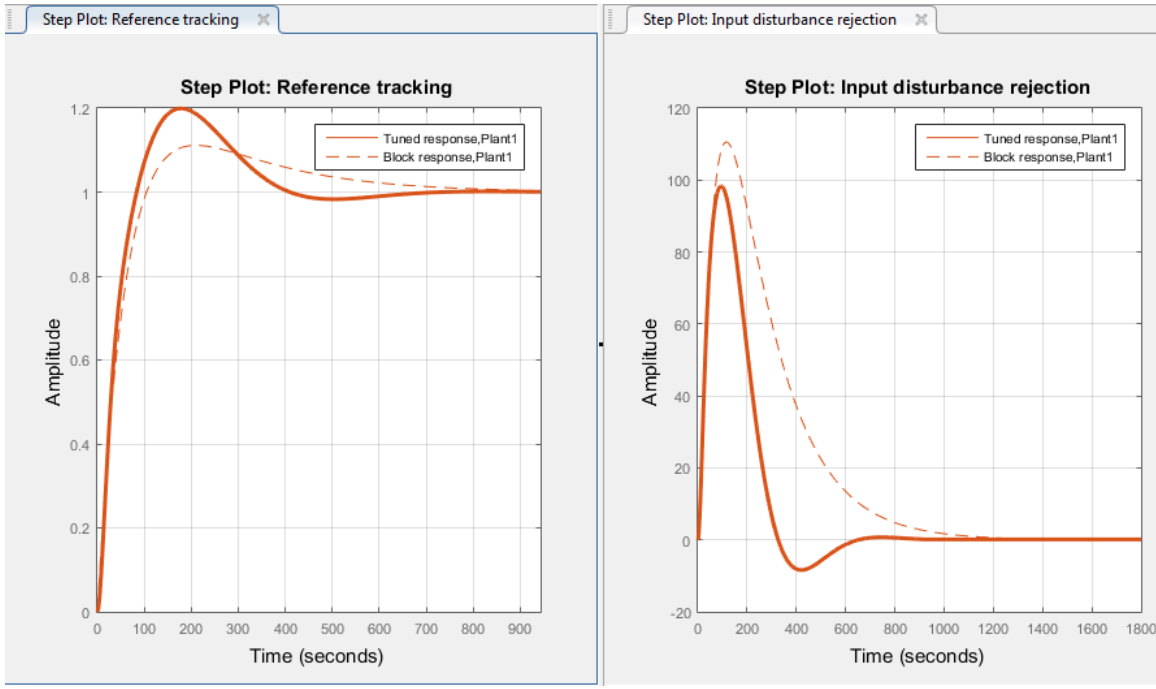


PID Tuner automatically retunes the controller coefficients with a focus on reference-tracking performance.



The responses with the balanced controller are now displayed as the `Block response`, and the controller tuned with a focus reference-tracking is the `Tuned response`. The plots show that the resulting controller tracks the reference input with considerably less overshoot and a faster settling time than the balanced controller design. However, the design yields much poorer disturbance rejection.

Finally, change the design focus to favor disturbance rejection. In the **Options** dialog box, in the **Focus** menu, select `Input disturbance rejection`.



This controller design yields improved disturbance rejection, but results in some increased overshoot in the reference-tracking response.

When you use design focus option, you can still adjust the **Transient Behavior** slider for further fine-tuning of the balance between these two measures of performance. Use the design focus and the sliders together to achieve the performance balance that best meets your design requirements. The effect of this fine-tuning on system performance depends strongly on the properties of your plant. For some plants, moving the **Transient Behavior** slider or changing the **Focus** option has little or no effect.

To obtain independent control over reference tracking and disturbance rejection, you can use a two-degree-of-freedom controller block, PID Controller (2DOF), instead of a single degree-of-freedom controller.

See Also

More About

- “PID Tuning Algorithm” on page 6-4
- “Analyze Design in PID Tuner” on page 6-9
- “Verify the PID Design in Your Simulink Model” on page 6-18
- “Design Two-Degree-of-Freedom PID Controllers” on page 6-35

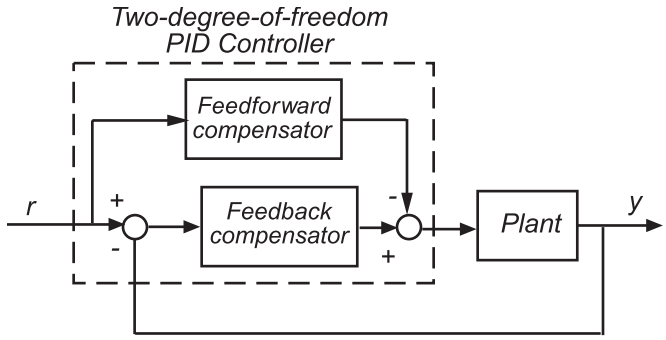
Design Two-Degree-of-Freedom PID Controllers

Use **PID Tuner** to tune two-degree-of-freedom PID Controller (2DOF) blocks to achieve both good setpoint tracking and good disturbance rejection.

In this section...

About Two-Degree-of-Freedom PID Controllers

A two-degree-of-freedom PID compensator, commonly known as an ISA-PID compensator, is equivalent to a feedforward compensator and a feedback compensator, as shown in the following figure.



The feedforward compensator is PD and the feedback compensator is PID. In the PID Controller (2DOF) block, the setpoint weights `b` and `c` determine the strength of the proportional and derivative action in the feedforward compensator. See the PID Controller (2DOF) block reference page for more information.

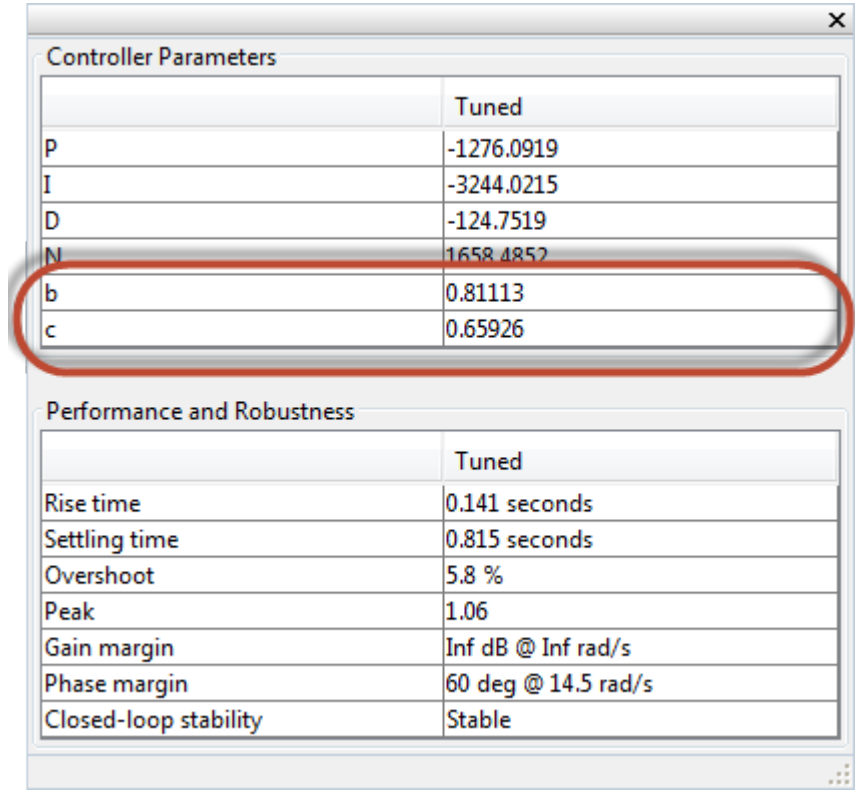
Tuning Two-Degree-of-Freedom PID Controllers

PID Tuner tunes the PID gains `P`, `I`, `D`, and `N`. For the PID Controller (2DOF) block, the tuner also automatically tunes the setpoint weights `b` and `c`. You can use the same techniques to refine and analyze the design that you use for tuning one-degree-of-freedom PID controllers.

To tune a PID Controller (2DOF) block in a Simulink model:

1 Double-click the block. In the block parameters dialog box, click **Tune**.

PID Tuner opens, linearizes the model at the model initial conditions, and automatically computes an initial controller design that balances performance and robustness. In this design, **PID Tuner** adjusts the setpoint weights `b` and `c` if necessary, as well as the PID gains. To see the tuned values of all coefficients, click **Show Parameters**.

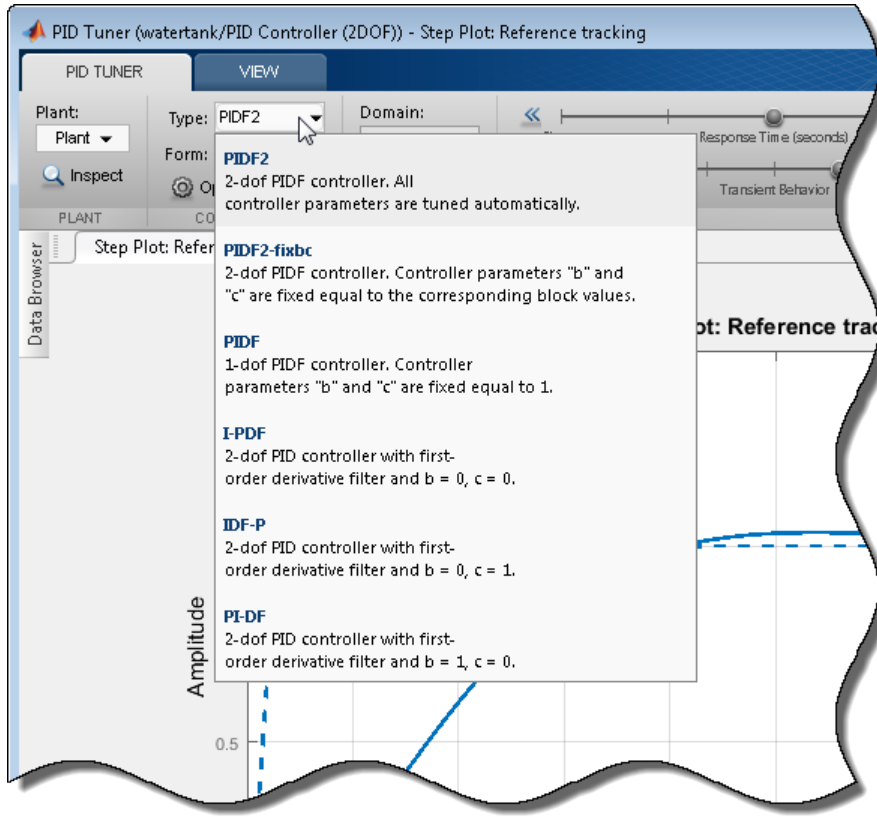


2 Analyze and refine the initial design, described in “Analyze Design in PID Tuner” on page 6-9. All the same response plots, design adjustments, and options are available for tuning 2DOF PID controllers as in the single-degree-of-freedom case.

3 Verify the controller design, as described in “Verify the PID Design in Your Simulink Model” on page 6-18.

Fixed-Weight Controller Types

When you tune a PID Controller (2DOF) block in **PID Tuner**, additional options for specifying the controller type become available in the **Type** menu. These options include controllers with fixed setpoint weights, such as the controllers described in “Specify PI-D and I-PD Controllers” on page 6-43.



The availability of some type options depends on the **Controller** setting in the PID Controller (2DOF) block dialog box.

Type	Description	Controller setting in block
`PIDF2`	2-DOF PID controller with filter on derivative term. **PID Tuner** tunes all controller parameters, including setpoint weights.	`PID`
`PIDF2-fixbc`	2-DOF PID controller with filter on derivative term. **PID Tuner** fixes setpoint weights at the values in the PID Controller (2DOF) block.	`PID`
`PIDF`	2-DOF controller with action equivalent to a 1-DOF PIDF controller, with fixed `b = 1` and `c = 1`.	`PID`
`I-PDF`	2-DOF PID controller with filter on derivative term, with fixed `b = 0` and `c = 0`.	`PID`
`IDF-P`	2-DOF PID controller with filter on derivative term, with fixed `b = 0` and `c = 1`.	`PID`
`PI-DF`	2-DOF PID controller with filter on derivative term, with fixed `b = 1` and `c = 0`.	`PID`
`PI2`	2-DOF PI controller. **PID Tuner** tunes all controller parameters, including setpoint weight on proportional term, `b`.	`PI`

Type	Description	Controller setting in block
`PI2-fixbc`	2-DOF PI controller with filter on derivative term. **PID Tuner** fixes setpoint weight `b` at the value in the PID Controller (2DOF) block.	`PI`
`PI`	2-DOF controller with action equivalent to a 1-DOF PI controller, with fixed `b = 1`.	`PI`
`PDF2`	2-DOF PD controller with filter on derivative term (no integrator). **PID Tuner** tunes all controller parameters, including setpoint weights.	`PD`
`PDF2-fixbc`	2-DOF PD controller with filter on derivative term. **PID Tuner** fixes setpoint weights at the values in the PID Controller (2DOF) block.	`PD`
`PD`	2-DOF controller with action equivalent to a 1-DOF PD controller, with fixed `b = 1` and `c= 1`.	`PD`

See Also

More About

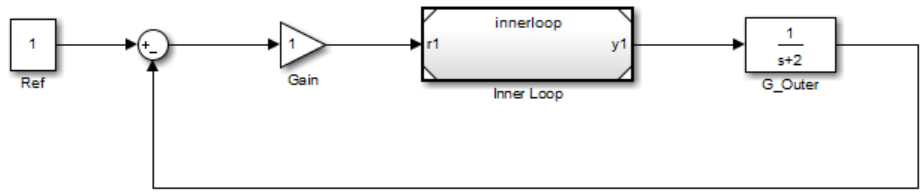
- “Analyze Design in PID Tuner” on page 6-9
- “Verify the PID Design in Your Simulink Model” on page 6-18
- “Specify PI-D and I-PD Controllers” on page 6-43

Tune PID Controller Within Model Reference

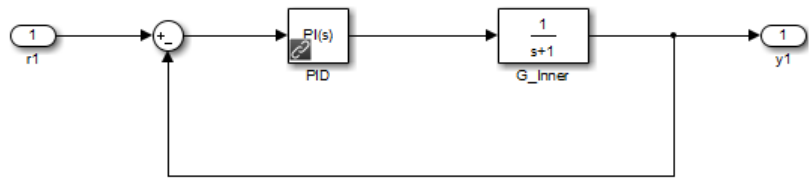
In Simulink, you can include one model inside another using model referencing (see "Overview of Model Referencing" (Simulink)). When using **PID Tuner** or **Frequency Response Based PID Tuner** to tune a PID Controller block in a referenced model, there are some constraints to be aware of.

In general, you can tune a PID Controller block in a referenced model using either **PID Tuner** or **Frequency Response Based PID Tuner**. When you open either tuner, the software prompts you to specify which model to use as the top-level model for linearization and tuning (**PID Tuner**) or estimation and tuning (**Frequency Response Based PID Tuner**). For example, consider the model `model_ref_pid`.

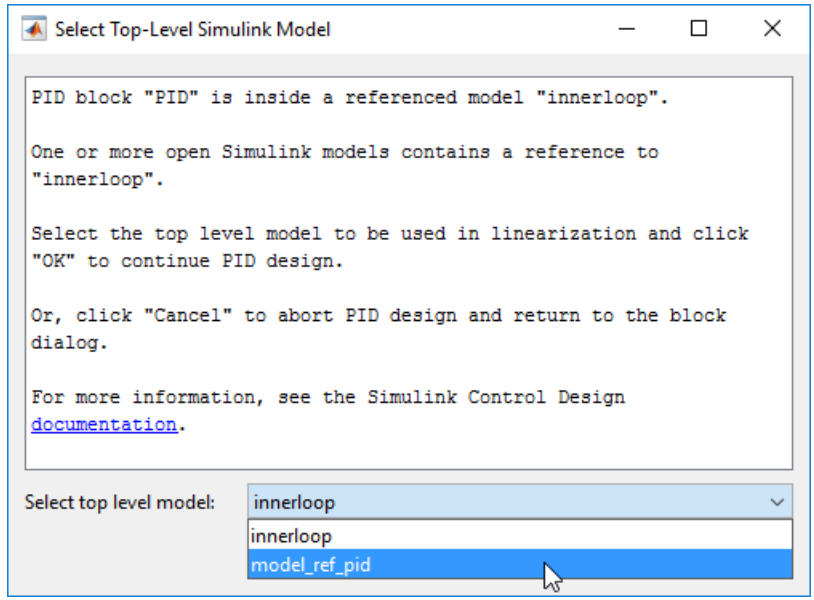
```
open('model_ref_pid');
```



The block `Inner Loop` is a referenced model that contains the PID Controller block to tune. Open the referenced model.



`Inner Loop` contains a PID controller block, `PID`. Open that block. In the **Select Tuning Method** drop-down list, select `Transfer Function Based (PID Tuner App)`, and click **Tune** to open **PID Tuner**. The software prompts you to select which open model is the top-level model for linearization and tuning. (Selecting `Frequency Response Based` to open **Frequency Response Based PID Tuner** results in a similar prompt.)



The available choices for top-level model include the referenced model itself, plus any open model in which the referenced model:

- Appears exactly once, and
- Is configured for normal simulation mode.

The tuning tools do not detect models that contain the model reference but are not open.

Selecting `innerloop` causes the tuner to disregard `model_ref_pid`. Instead, the tuner tunes the `PID Controller` block for the plant `G_Inner` alone, as if there were no outer loop.

Alternatively, you can select `model_ref_pid` as the top-level model. When you do so, the tuner considers the dynamics of both the inner and outer loops, and tunes with both loops closed. In this case, PID controller sees the effective plant `(1+G_Outer*Gain)*G_Inner`.

Select the desired top-level model, and click **OK**. The tuner you selected with the **Select Tuning Method** opens for tuning the specified top-level model.

Models with Multiple Instances of the Referenced Model

Sometimes, tuning can proceed when the referenced model appears multiple times in an open model. If the following conditions are met, you can tune the PID Controller block, using the referenced model as the top-level model:

- The only open models that contain the model reference have multiple instances of it, and
- At least one of these instances is in normal mode.

When this condition occurs, the software issues a warning. In this case, because the tuner can only tune with respect to the referenced model, you cannot specify a top-level model.

Referenced Model in Accelerated or Other Simulation Modes

If there is no normal mode instance of the referenced model in any open model, tuning cannot proceed. In this case, the software issues an error. To tune the PID Controller block, convert some instance of the referenced model in an open model to normal simulation mode.

See Also

More About

- “Overview of Model Referencing” (Simulink)
- “Choosing a Simulation Mode” (Simulink)
- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3

Specify PI-D and I-PD Controllers

In this section...

About PI-D and I-PD Controllers

PI-D and I-PD controllers are used to mitigate the influence of changes in the reference signal on the control signal. These controllers are variants of the 2DOF PID controller.

The general formula of a parallel-form 2DOF PID controller is:

$$u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y).$$

Here, r and y are the reference input and measured output, respectively. u is the controller output, also called the control signal. P, I, and D specify the proportional, integral, and derivative gains, respectively. N specifies the derivative filter coefficient. b and c specify setpoint weights for the proportional and derivative components, respectively. For a 1DOF PID, b and c are equal to 1.

If r is nonsmooth or discontinuous, the derivative and proportional components can contribute large spikes or offsets in u, which can be infeasible. For example, a step input can lead to a large spike in u because of the derivative component. For a motor actuator, such an aggressive control signal could damage the motor.

To mitigate the influence of r on u, set b or c, or both, to 0. Use one of the following setpoint-weight-based forms:

- PI-D (b = 1 and c = 0) — Derivative component does not directly propagate changes in r to u, whereas the proportional component does. However, the derivative component, which has a greater impact, is suppressed. Also referred to as the derivative of output controller.

  The general formula for this controller form is:

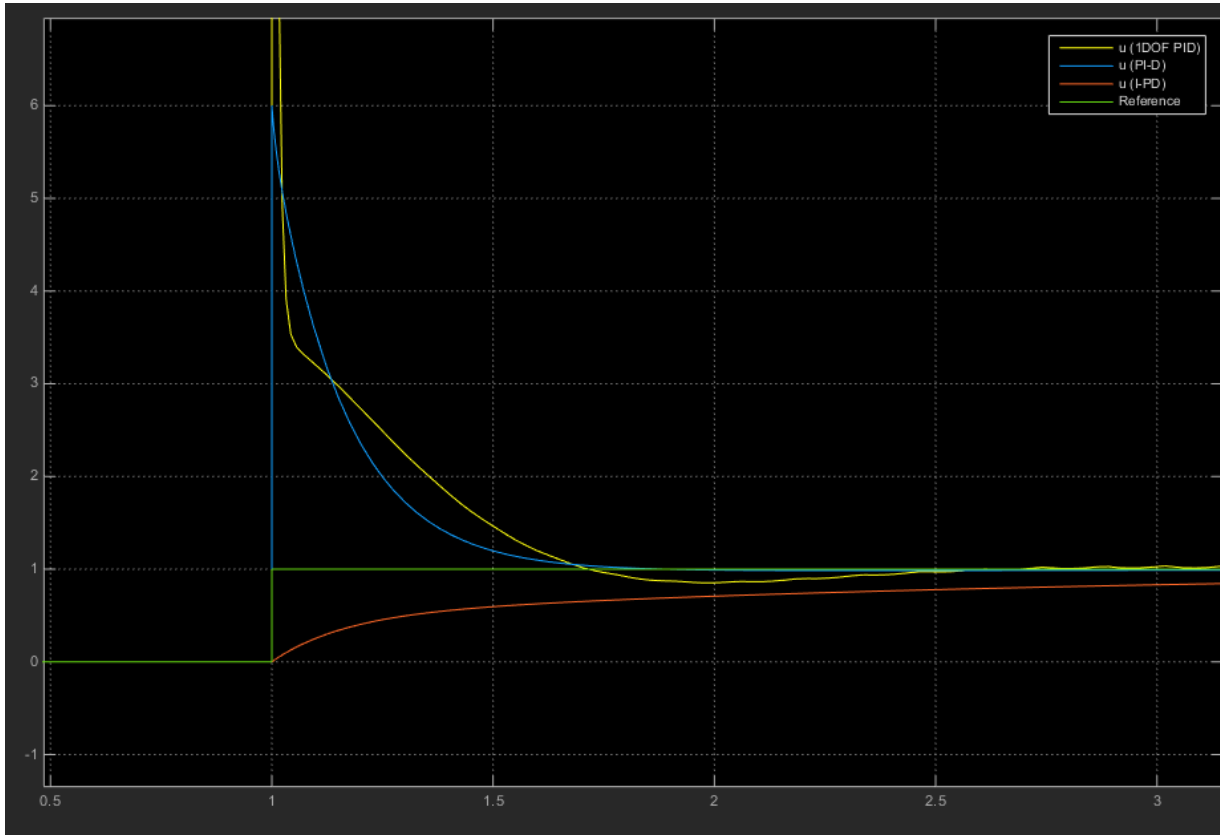
$$u = P(r - y) + I\frac{1}{s}(r - y) - D\frac{N}{1 + N\frac{1}{s}}y.$$

- I-PD ($b = 0$ and $c = 0$) — Proportional and derivative components do not directly propagate changes in r to u.

  The general formula for this controller form is:

  $$u = -Py + I\frac{1}{s}(r - y) - D\frac{N}{1 + N\frac{1}{s}}y.$$

The following plot shows u for different PID forms for a step reference. The 1DOF PID controller results in a large spike when the reference changes from 0 to 1. The PI-D form results in a smaller jump. In contrast, the I-PD form does not react as much to the change in r.



You can tune the *P*, *I*, *D*, and *N* coefficients of a PI-D or I-PD controller to achieve the desired disturbance rejection and reference tracking.

Specify PI-D and I-PD Controllers Using PID Controller (2DOF) Block

To specify a PI-D or I-PD Controller using the PID Controller (2DOF) block, open the block dialog. In the **Controller** menu, select `PID`.

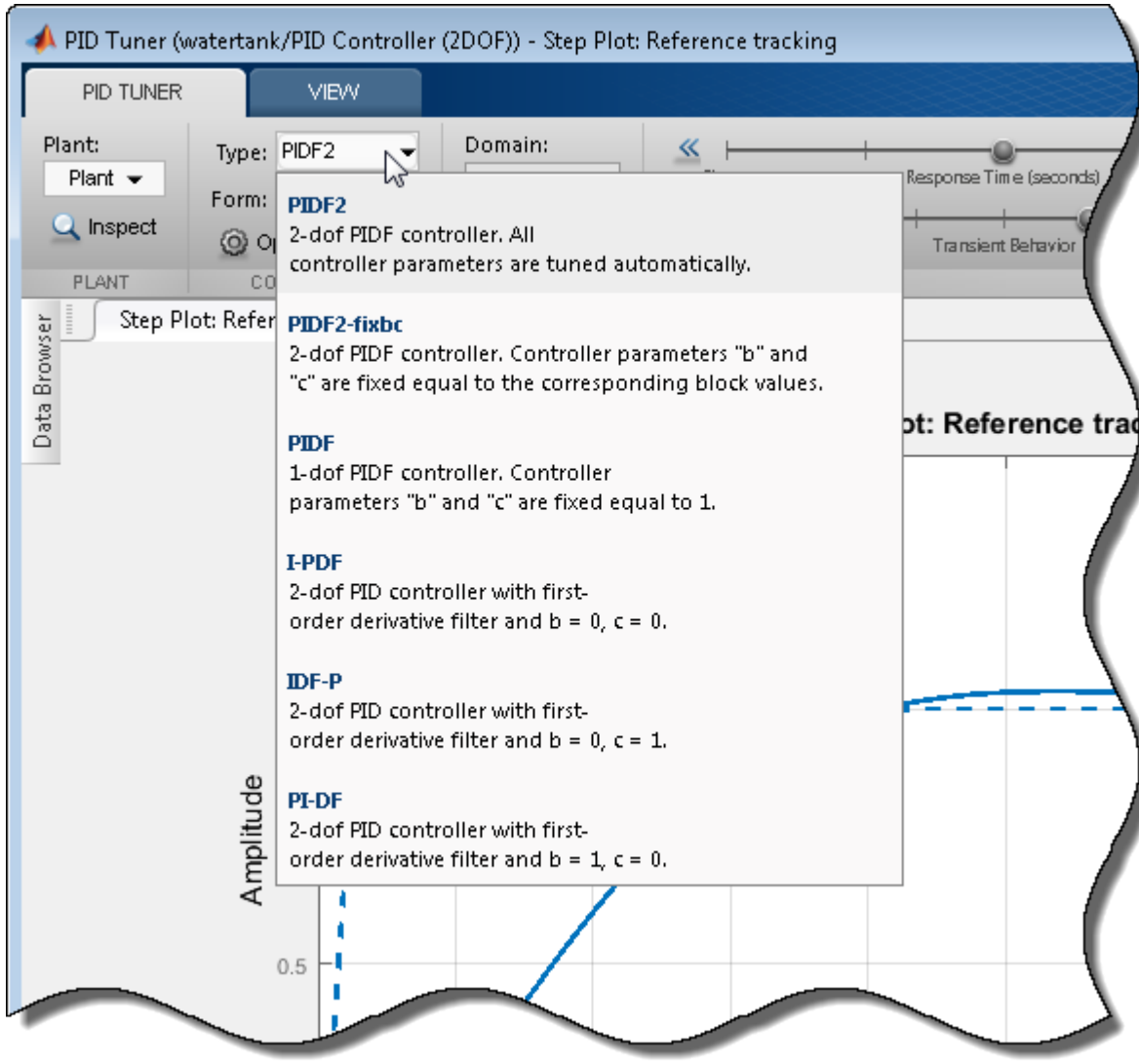
- For a PI-D controller, enter `1` in the **Setpoint weight (b)** box, and `0` in the **Setpoint weight (c)** box.
- For an I-PD controller, enter `0` in the **Setpoint weight (b)** box, and `0` in the **Setpoint weight (c)** box.

For an example that demonstrates the PI-D and I-PD controller forms, type `ex_scd_pid2dof_setpoint_based_controllers`. This opens a model that compares the performance of a 1DOF PID, a PI-D, and an I-PD controller.

Automatic Tuning of PI-D and I-PD Controllers

You can use **PID Tuner** to automatically tune PI-D and I-PD controllers while preserving the fixed b and c values. To do so:

1. In the model, double-click the PID Controller (2DOF) block. In the block dialog box, in the **Controller** menu, select `PID`.
2. Click **Tune**. **PID Tuner** opens.
3. In **PID Tuner**, in the **Type** menu, select `PI-DF` or `I-PDF`. **PID Tuner** retunes the controller gains, fixing $b = 1$ and $c = 0$ for PI-D, and $b = 0$ and c for I-PD.



You can now analyze system responses as described in “Analyze Design in PID Tuner” on page 6-9.

See Also

PID Controller | PID Controller (2 DOF)

More About

- "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection" on page 6-23
- "Design Two-Degree-of-Freedom PID Controllers" on page 6-35

Design PID Controller from Plant Frequency-Response Data

Most Simulink Control Design PID tuning tools design PID gains based on a linearized plant model. When your plant model does not linearize or linearizes to zero, one option is to design a PID controller based on simulated frequency-response data. Simulink Control Design gives you several ways to do so.

Use Frequency Response Based PID Tuner

Use **Frequency Response Based PID Tuner** to design a PID controller using estimated plant frequency responses near the target open-loop bandwidth. Advantages of this approach include:

- **Frequency Response Based PID Tuner** works even if disturbances are present in the plant model.
- You can configure the estimation and tuning in one dialog box, making tuning less complex than using `frestimate` or Linear Analysis Tool to estimate the frequency response.

For more information about using **Frequency Response Based PID Tuner**, see "Frequency Response Based Tuning Basics" on page 6-51.

Use frestimate or Linear Analysis Tool

Use the `frestimate` command or the frequency-response estimation workflow in Linear Analysis Tool to estimate the plant frequency response over a range of frequencies that you specify. This approach results in a frequency-response data (`frd`) model object that you then import into **PID Tuner**. Advantages of this approach include:

- You do not have to specify a control bandwidth ahead of time. **PID Tuner** chooses an initial control bandwidth, which you can adjust to achieve the desired balance between performance and robustness.
- You can use the interactive tuning and analysis tools of **PID Tuner** to examine the estimated linear response of the tuned system in the frequency domain. Also, you can use the `frd` model of the plant for other analysis tasks.
- Depending on the particulars of your model, this approach can be faster, because **Frequency Response Based PID Tuner** simulates your model twice.

For more information, see:

- “Design PID Controller Using FRD Model Obtained From "frestimate" Command” on page 6-102
- “Estimate Frequency Response Using Linear Analysis Tool” on page 5-26

See Also

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3
- “Frequency Response Based Tuning Basics” on page 6-51
- “Design PID Controller Using FRD Model Obtained From "frestimate" Command” on page 6-102

Frequency Response Based Tuning Basics

Frequency Response Based PID Tuner simulates the model to estimate the plant frequency responses at a few frequencies near the control bandwidth. It then uses the estimated frequency response to tune the gains in your PID Controller block. This tuner is a useful alternative when **PID Tuner** cannot linearize the plant at the operating point you want to use for tuning.

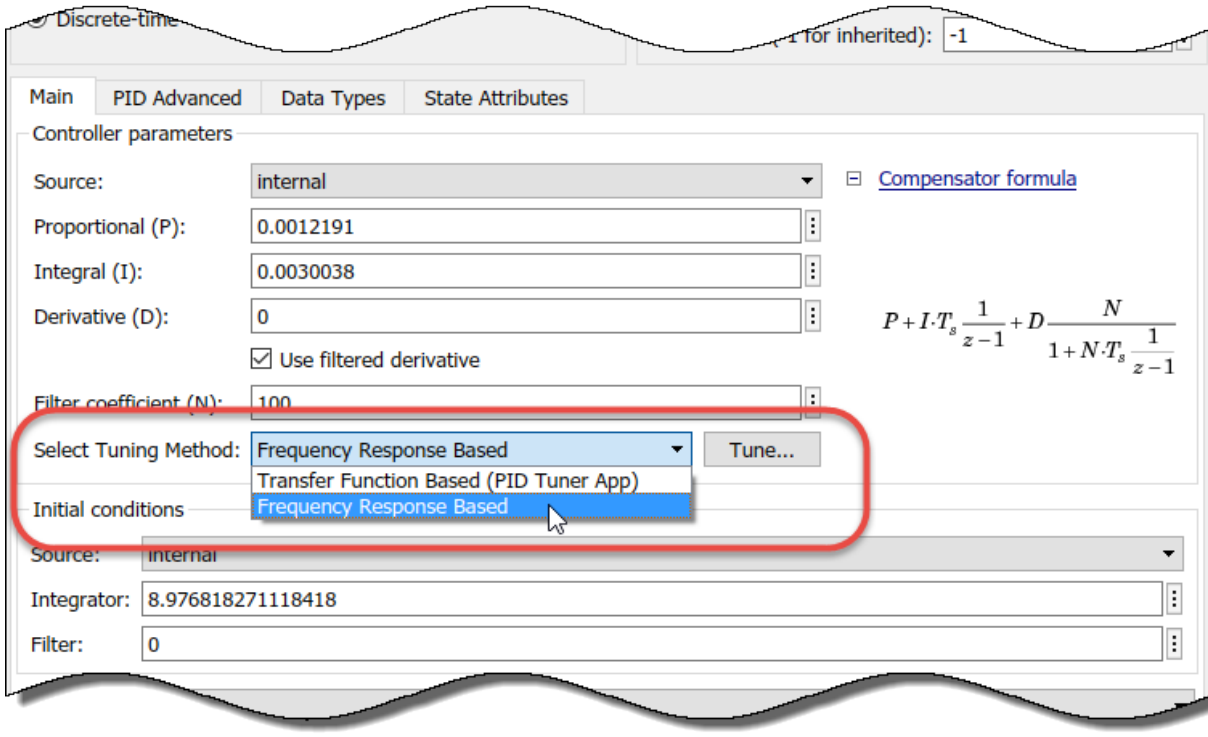
How Frequency Response Based PID Tuner Works

Like the interactive **PID Tuner**, the **Frequency Response Based PID Tuner** considers the plant to be all blocks in the loop between the PID Controller block output and input. The **Frequency Response Based PID Tuner** performs a perturbation experiment to estimate the open-loop frequency response of the plant. To do so, the tuner performs the following steps:

1. Simulates the model to generate a baseline plant response, including plant input and output values at the nominal operating point.
2. Breaks the loop at the controller output and simulates the model again, applying perturbation signals to the plant. The perturbations include sinusoidal signals at frequencies $[1/3,1,3,10]\omega_c$, where ω_c is the target bandwidth you specify for tuning. If the plant is asymptotically stable, the applied signal also includes a step perturbation.
3. Measures the response to the perturbation at the controller input.
4. Takes the difference between the two simulations, removing any effects of disturbances in the model.
5. Uses the resulting data to estimate the plant frequency response at the four frequencies. For asymptotically stable plants, the tuner uses the response to the step perturbation to estimate the plant DC gain.
6. Uses the estimated frequency response to compute PID gains that balance performance and robustness.

Open Frequency Response Based PID Tuner

To open the **Frequency Response Based PID Tuner**, in the PID Controller block dialog box, in the **Select Tuning Method** drop-down list, select `Frequency Response Based`.



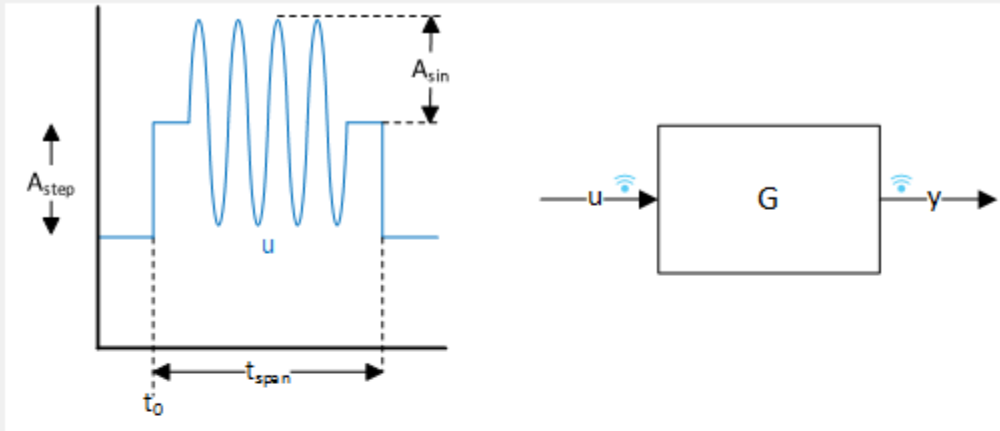
Click **Tune**. The **Frequency Response Based PID Tuner** opens. The tuner reads some parameters from the PID Controller block. These parameters include:

- Controller type (such as PI, PD, or PID)
- Controller form (parallel or ideal)
- Controller time domain (continuous-time or discrete-time)
- Controller sample time

In the **Frequency Response Based PID Tuner**, You configure the settings for the estimation experiment and the tuning goals.

Frequency Response Based PID Tuner

Description



When you click Tune, two rounds of simulation run to:
(1) Perturb open-loop plant with sine and step signals during the specified time frame
(2) Estimate plant frequency responses and dc gain from the experiment
(3) Tune PID gains to achieve the target phase margin at the target bandwidth

Experiment Settings

Plant is asympototically stable — Plant has a single integrator

Start time (t0):	0	Duration (tspan):	100
Sine amplitudes (Asin):	1	Step amplitude (Astep):	1

Design Specifications

Target bandwidth (rad/sec):	1	Target phase margin (degrees):	60

Automatically update block | Tune | Cancel

Tuning Results

Gains: PID gains are not tuned yet.
Click "Tune" to start tuning.

Update PID Block | Export To MATLAB | Hide | Close | Help

Configure Experiment Settings

In the **Experiment Settings** section, you specify parameters that control the frequency-response estimation experiment. For more details about these settings, click **Help**.

1 Specify whether the plant is asymptotically stable or has a single integrator. If the plant is asymptotically stable, the estimation experiment includes an estimation of the plant DC gain. The **Frequency Response Based PID Tuner** performs this estimation by injecting a step signal into the plant.

Caution Do not use the **Frequency Response Based PID Tuner** with an unstable plant or a plant containing multiple integrators.

---

2 Specify the start time of the experiment in the **Start time (t0)** field. Start the experiment when the plant is at the desired equilibrium operating point. For instance, if you know that your simulation must run to 10 s for the plant to reach such an operating point, specify a start time of 10.

3 Specify the experiment duration in the **Duration (tspan)** field. Let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning that you specify.

4 Specify the perturbation amplitudes. During the tuning experiment, the **Frequency Response Based PID Tuner** injects a sinusoidal signal into the plant at four frequencies, $[1/3,1,3,10]\omega_c$. Use the **Sine amplitudes (Asin)** field to specify the amplitudes of these injected signals. You can provide a scalar value to inject the same amplitude at each frequency, or a vector of length 4 to specify different amplitudes for each.

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the four frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that there is a sharp decay in response over the range of frequencies, consider decreasing the amplitude of $(1/3)\omega_c$ and increasing the amplitude of $10\omega_c$. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level.

- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output.

In the experiment, the four sinusoidal signals are superimposed (with the step perturbation, if any). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

- Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
- How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID controller used in engine-speed control. You have determined that a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

If your plant is asymptotically stable, specify amplitude of the step perturbation in the **Step amplitudes (Astep)** field. The considerations for choosing a step amplitude are the same as the considerations for specifying the step amplitudes.

Configure Design Goals

In the **Design Specifications** section of the dialog box, you specify your goals for PID tuning.

Specify the target bandwidth in the **Target bandwidth (rad/sec)** field. The target bandwidth is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ, a good guess for the target bandwidth is $2/\tau$.

In the **Target phase margin (degrees)** field, specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

For more details about these settings, click **Help**.

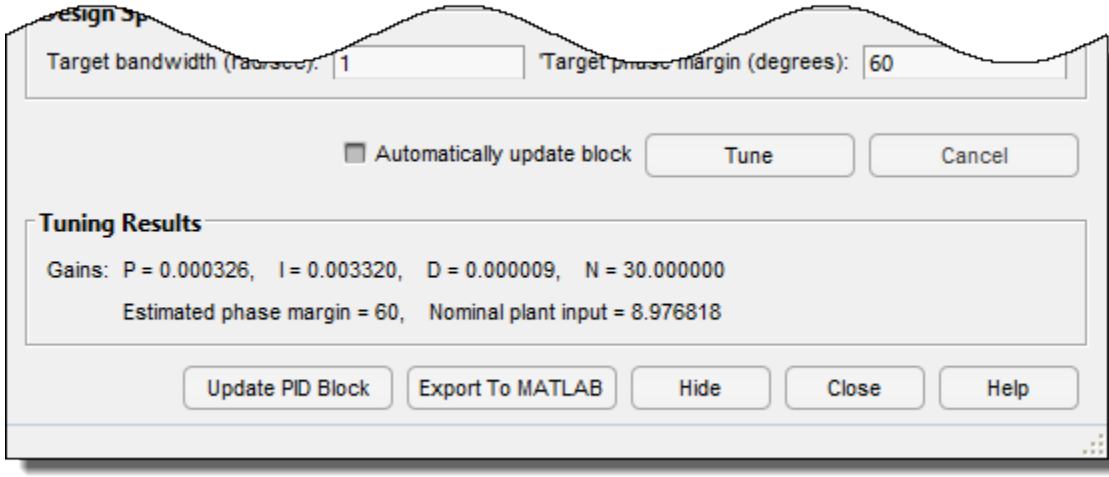
Tune and Validate Controller Gains

Click **Tune** to initiate the frequency-response estimation experiment. While the estimation experiment is running, the tuner:

- Closes the open PID Controller block dialog.
- Clears any previous tuning results displayed in the tuner dialog box.
- Replaces the PID Controller block in your model with an unnamed subsystem.

Note When the estimation experiment is completed or canceled, the tuner restores the PID Controller block. This process might result in some displacement of signal wires on the model canvas, and puts your Simulink model in a state with unsaved changes.

When the estimation experiment ends, the tuner computes new PID gains and displays them in the **Tuning Results** section of the dialog box. (For more information about the tuning results, click **Help**.)



If **Automatically update block** is selected, the **Frequency Response Based PID Tuner** writes the new PID gains to the PID Controller block when tuning is completed. Otherwise, click **Update PID Block** to write the tuned gains to the block. Simulate the model to validate the tuned gains against your full nonlinear system.

For an example illustrating the use of the **Frequency Response Based PID Tuner** to tune a PID Controller block in a Simulink model that does not linearize, see “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 6-58.

See Also

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3
- “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 6-58
- “Design PID Controller Using FRD Model Obtained From "frestimate" Command” on page 6-102

Design PID Controller Using Plant Frequency Response Near Bandwidth

This example shows one of several ways to tune a PID controller for plants that cannot be linearized. In this example, you use the Frequency Response Based PID Tuner to automatically characterize the frequency response of a buck converter around the control bandwidth, and then tune the PID controller.

Buck Converter Model

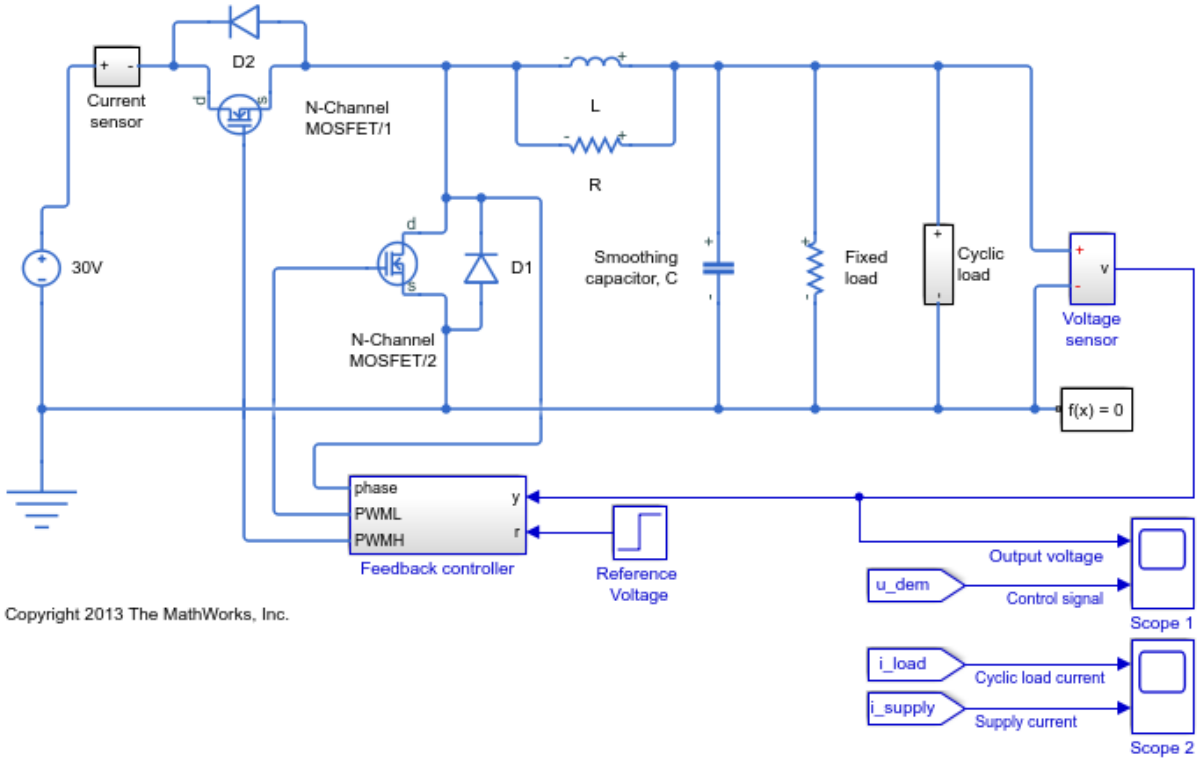
Buck converters convert DC to DC. The model in this example uses a switching power supply to convert a 30V DC supply into a regulated DC supply. The converter is modeled using MOSFETs rather than ideal switches to ensure that device on-resistances are correctly represented. The converter response from reference voltage to measured voltage includes the MOSFET switches. Traditional PID design requires a linear model of the system from "the reference voltage" (controller output) to measured voltage. Here, however, because of the switches, automated linearization results in a zero system. When a model linearizes to zero, several alternatives are available:

- **Re-linearize the system**. Linearize the model at a different operating point or simulation snapshot time.
- **Identify a new plant**. Use measured or simulated data to identify a plant model (requires System Identification Toolbox).
- **Frequency response based tuning**. Use simulated data to obtain the frequency response for the plant.

For this example, use the **Frequency Response Based PID Tuner** to estimate the frequency responses of the system and tune the PID controller. For an example that uses system identification to identify a plant model, see “Design a PID Controller Using Simulated I/O Data”.

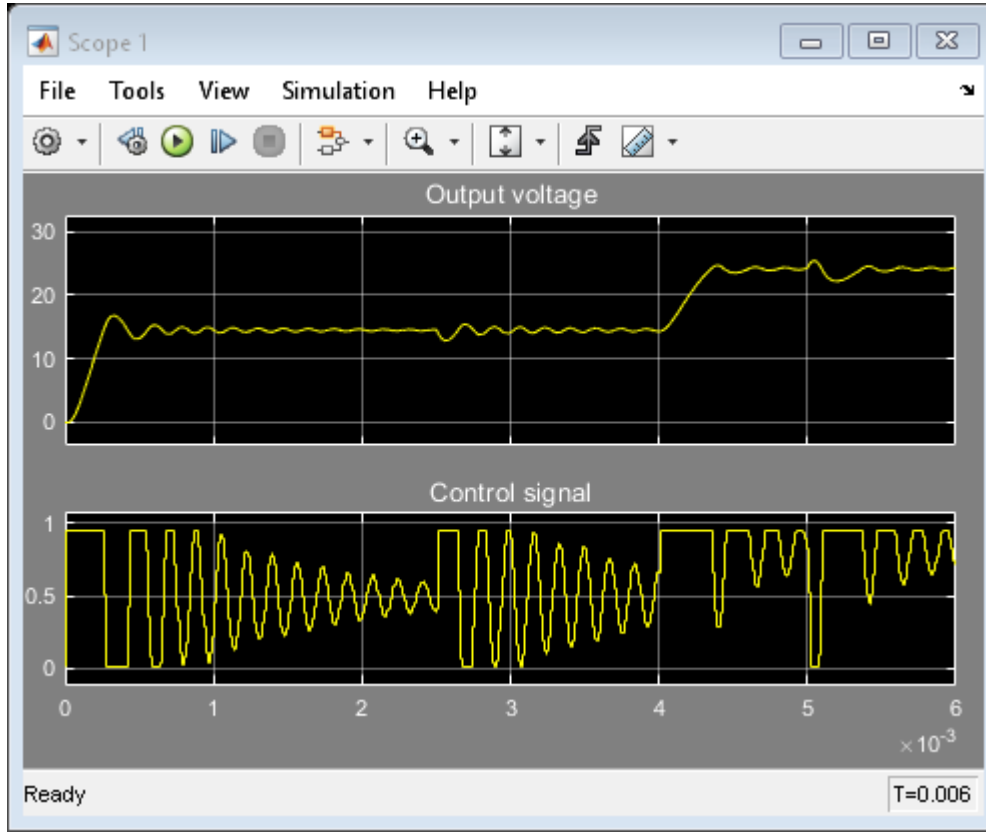
The buck converter model is described in more detail in the Simscape Electronics example `elec_switching_power_supply`.

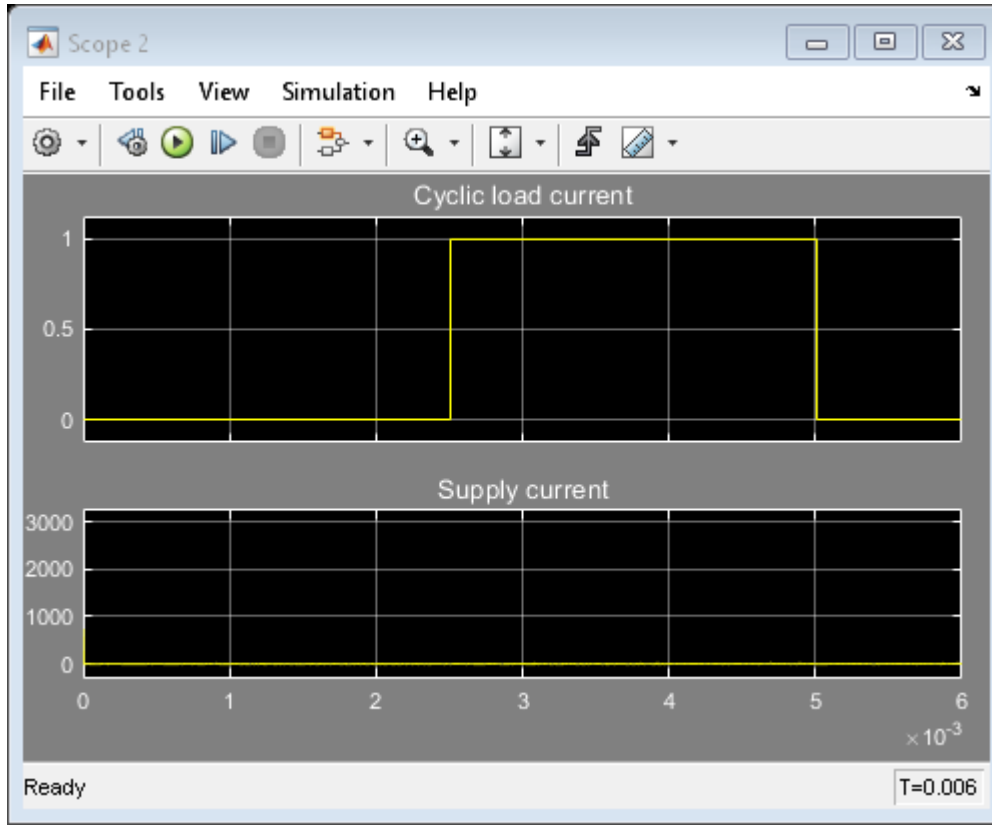
```
open_system('scdbuckconverter')
```



The model is configured with a reference voltage that switches from 15 to 25 Volts at 0.004 seconds and a load current that is active from 0.0025 to 0.005 seconds. The controller is initialized with default gains and results in overshoot and slow settling time. Simulating the model shows the underdamped and slow response nature of the system.

```
sim('scdbuckconverter')
open_system('scdbuckconverter/Scope 1')
open_system('scdbuckconverter/Scope 2')
```


Scope 1
File
Tools
View
Simulation
Help
Output voltage
30
20
10
0
Control signal
1
0.5
0
0
1
2
3
4
5
6
×10-3
Ready
T=0.006



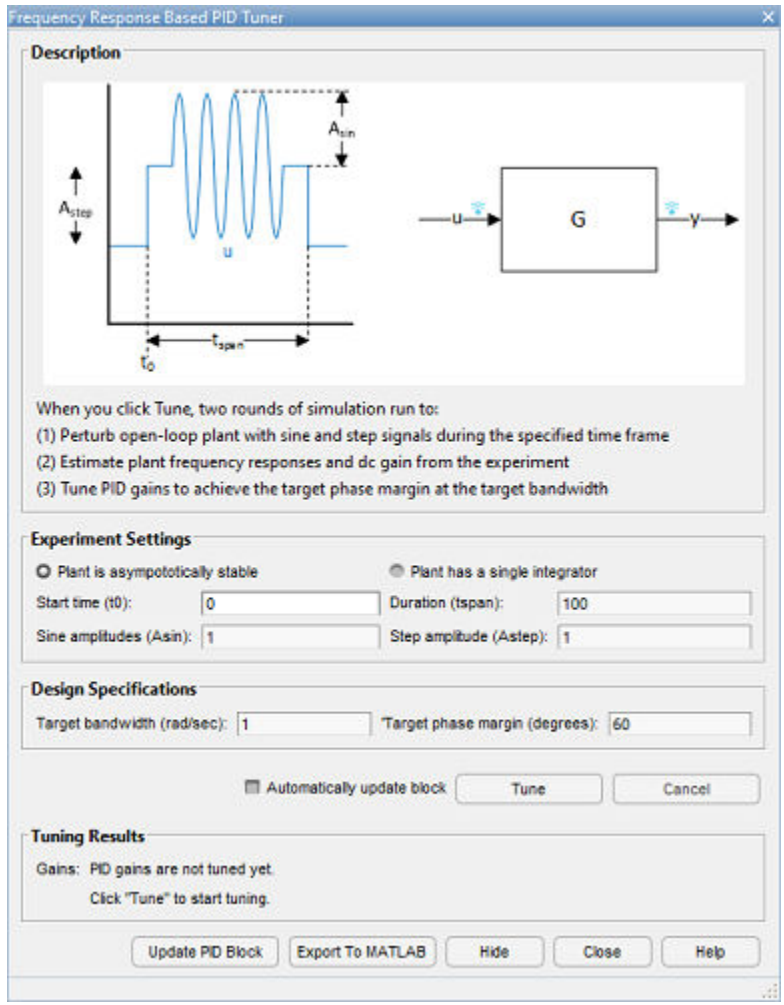
For this example, improve the bandwidth and phase margin of the system to achieve better performance by characterizing the system using frequency response estimation and tuning the PID gains. When tuning the PID controller note the following characteristics of the buck converter system:

- No system process or sensor noise
- Controller input is the PWM signal
- PWM signal is limited (saturated) to be between 0 and 1
- Nominal output of controller at steady-state is 0.5

For buck converter systems, it is desired to have a system with a low rise time and low overshoot. For this example, tune the controller to achieve a desired rise time of 250e-6 seconds and an overshoot of less than 10%.

Open Frequency Response Based PID Tuner

Open the **Feedback controller** subsystem and then open the **PID Controller** block dialog. In **Select Tuning Method**, select **Frequency Response Based** and click **Tune**. The **Frequency Response Based PID Tuner** opens for the buck converter controller.



The **Frequency Response Based PID Tuner** automatically tunes a PID controller for the plant using two simulations. The first simulation generates a baseline response. The

second simulation breaks the loop at the plant input, and perturbs the plant with sine and step signals. The tuner takes the difference between the two simulated responses, which removes the effect of any disturbances in the model. The tuner then uses the resulting data to estimate the plant frequency response. Finally, it uses the estimated frequency response to compute PID gains.

When you open the **Frequency Response Based PID Tuner**, it reads parameters from the PID Controller block to determine the structure of your PID controller. These parameters include:

- PID Controller Type (P, I, PI, PID etc.)
- PID Controller Form (Parallel, Ideal)
- Integrator Method, if applicable (Forward Euler, Trapezoidal etc.)
- Derivative Filter Method, if applicable (Forward Euler, Trapezoidal etc.)
- Sample Time, if applicable

Specify Experiment Settings

Before tuning, specify parameters of the experiement the tuner performs to estimate the frequency response of the plant.

Start time is the time, in seconds, at which the tuner begins applying the perturbation signals to the plant. Choose a start time at which the plant is at the nominal operating point you want to use for tuning. For this example, the buck converter has an initial transient that falls off by 0.002 seconds. Therefore, enter 0.002 for **Start Time**.

Specify the **Duration** of the perturbation experiment. A conversative estimate for the duration of the experiment is 100 divided by the target bandwidth. The target bandwidth is approximately $2/\tau$, where τ is the desired rise time. For this example, the desired rise time is 250e-6 seconds which results in a target bandwidth of 8000 radians per seconds In this example a conservative estimate for the duration would then be 100/8000 or 0.0125 seconds. Choose 0.0125 seconds for the **Duration**.

During the experiment, the tuner injects sinusoidal signals into the plant at four frequencies, [1/3, 1, 3, 10] ω_c, where ω_c is the target bandwidth you specify for tuning. Specify the amplitudes of the injected sine waves in the **Sine Amplitudes** field.

Choose amplitudes which have magnitudes above the noise floor of the system and will not saturate the system. For this example there is no noise in the system to consider. However, the controller output (duty cycle of the PWM) is limited to [0 1] and the

nominal output of the contrlller at steady-state is 0.5. To remain within these limits, specify a sine amplitude of 0.1. Specifying a scalar value uses the same amplitude for at all four frequencies.

For an asymptotically stable plant, the tuner also injects a step signal to estimate the plant DC gain. Choose an amplitude for this step signal based on the same considerations you used to choose teh sine amplitudes. For this example, enter 0.1 in the **Step Amplitude** field as well.

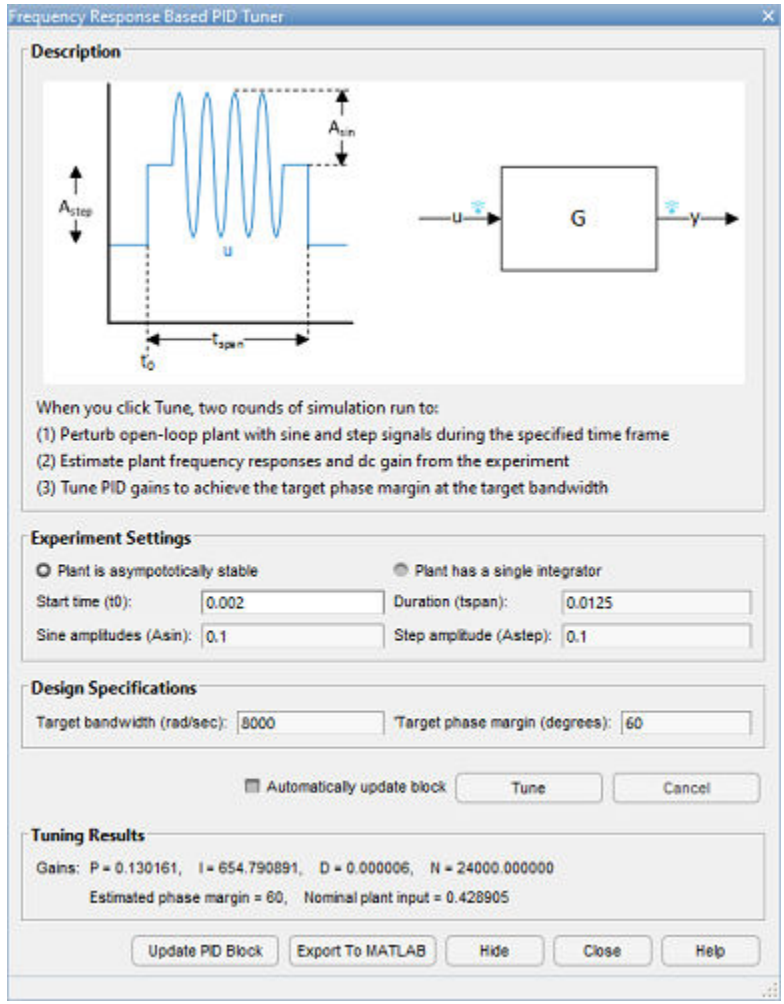
Specify Design Goals

Finally, specify the target bandwidth for tuning. As noted previously, the target bandwidth is 8000 radians per second. Enter 8000 in the **Bandwidth** field. The default target phase margin, 60 degrees, corresponds to an overshoot of about 10% or better.

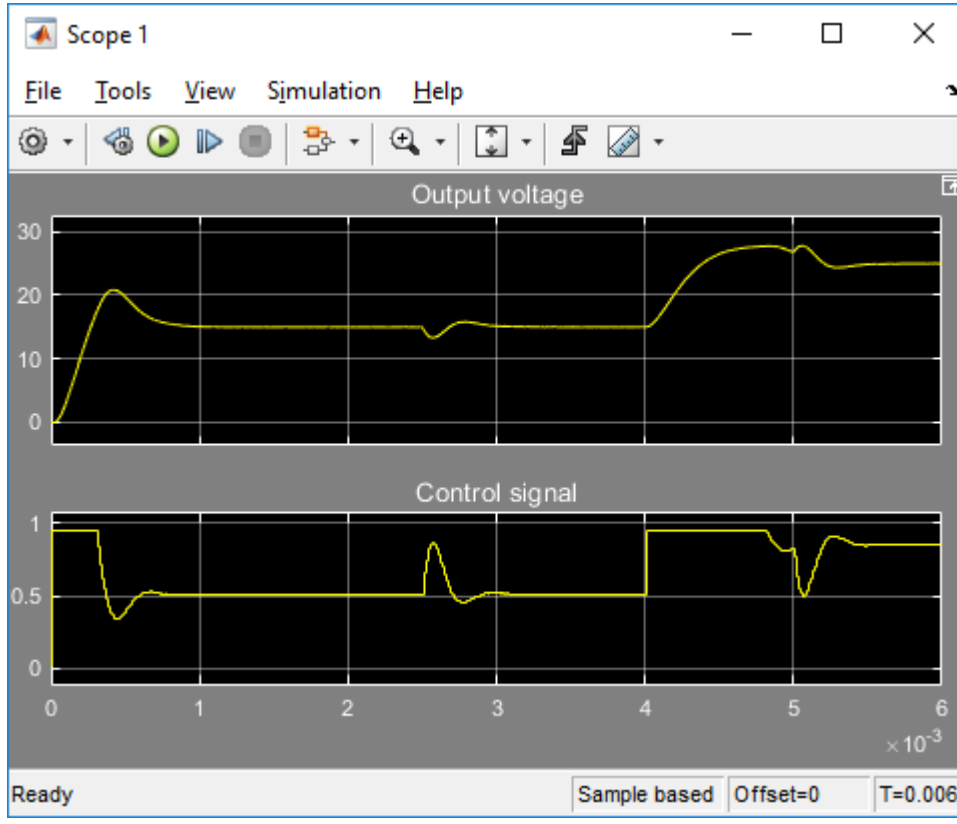
Tune the PID Controller and Validate the Results

Click **Tune** to begin the two simulations of the buck converter and tune the PID Controller.

At the conclusion of the tuning procedure the tuned gains, estimated phase margin and nominal plant input are displayed in **Frequency Response Based PID Tuner** dialog in the **Tuning Results** section. Check the estimated phase margin to ensure that it is close to the **Target phase margin**.



To verify the results simulate the model using the tuned PID gains. To do so, update the gains in the PID Controller block. Click **Update PID Block** to write the tuned gains to the PID Controller block. Then, simulate the model to confirm the PID controller performance.




```
bdclose('scdbuckconverter')
```

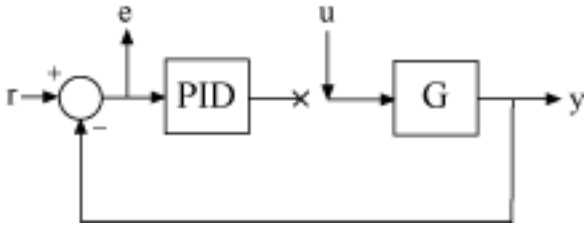
See Also

Import Measured Response Data for Plant Estimation

This example shows how to use **PID Tuner** to import measured response data for plant estimation.

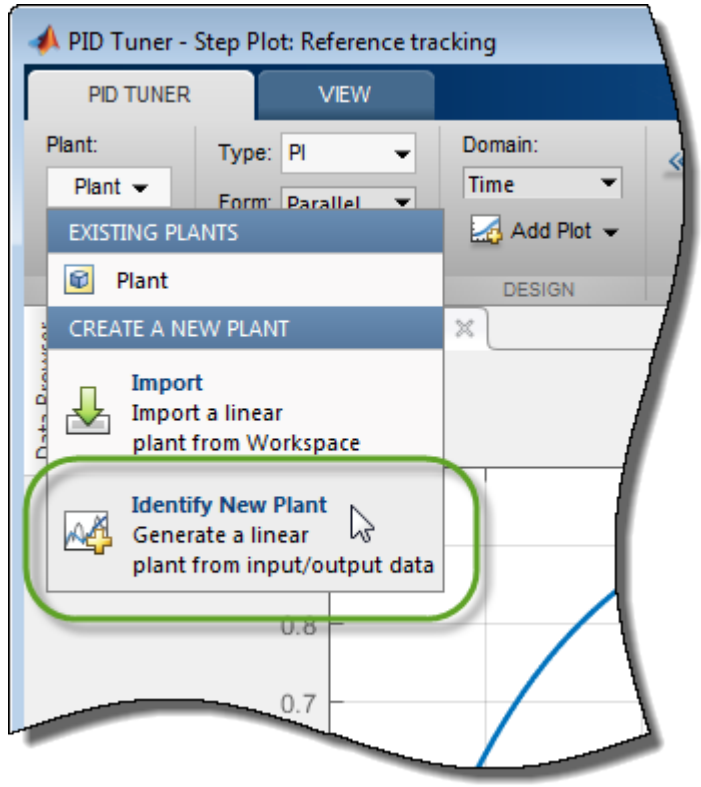
If you have System Identification Toolbox software, you can use **PID Tuner** to estimate the parameters of a linear plant model based on time-domain response data. **PID Tuner** then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink model. Plant estimation is especially useful when your Simulink model cannot be linearized or linearizes to zero.

When you import response data, **PID Tuner** assumes that your measured data represents a plant connected to the PID controller in a negative-feedback loop. In other words, **PID Tuner** assumes the following structure for your system. **PID Tuner** assumes that you injected an input signal at `u` and measured the system response at `y`, as shown.



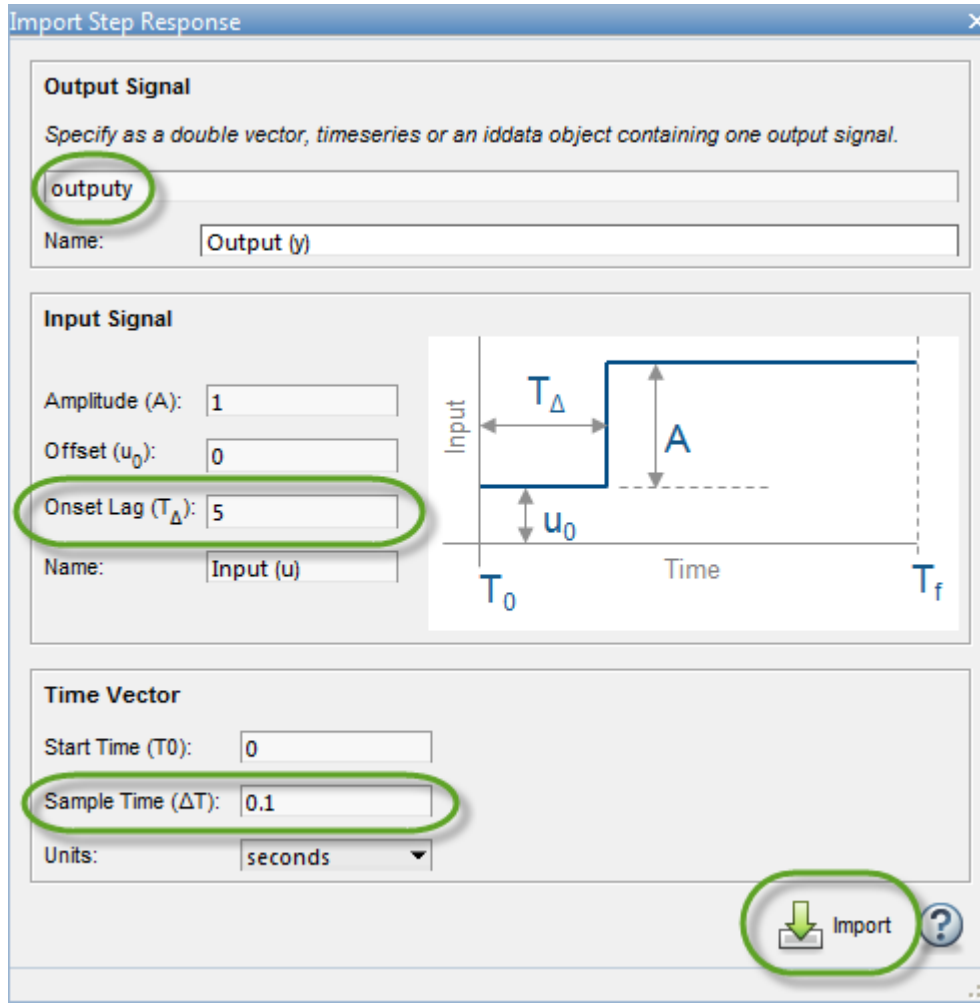
You can import response data stored in the MATLAB workspace as a numeric array, a `timeseries` object, or an `iddata` object. To import response data:

1. In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select `Identify New Plant`.

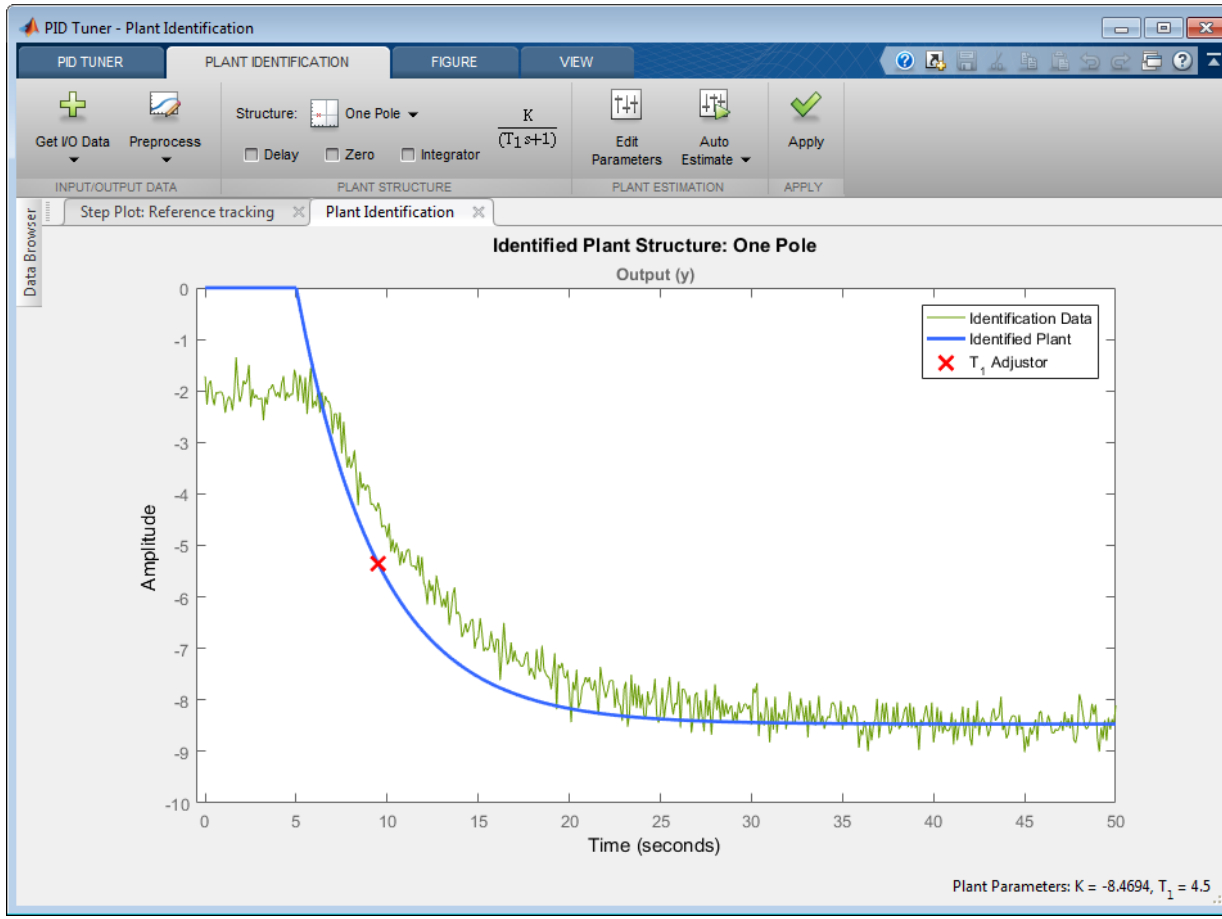


2 In the **Plant Identification** tab, click **Get I/O data**. Select the type of measured response data you have. For example, if you measured the response of your plant to a step input, select **Step Response**. To import the response of your system to an arbitrary stimulus, select **Arbitrary I/O Data**.

3 In the Import Response dialog box, enter information about your response data. For example, for step-response data stored in a variable `outputy` and sampled every 0.1s:



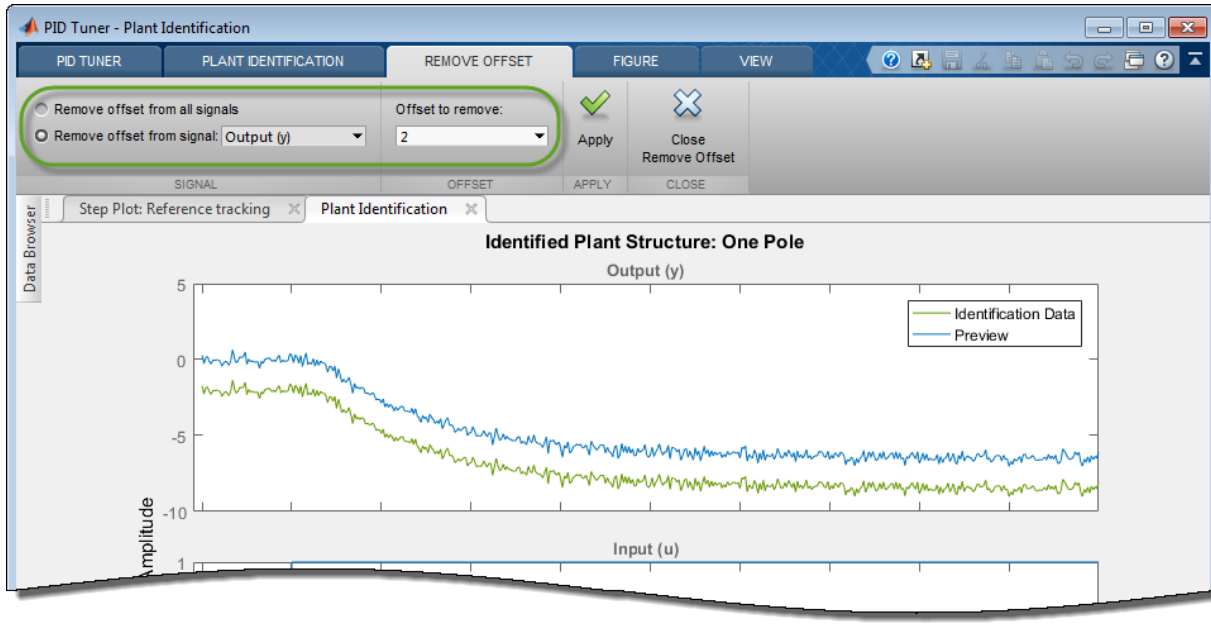
Click **Import**. The **Plant Identification** tab opens, displaying the response data and the response of an initial estimated plant.



4 Depending on the quality and features of your response data, you might want to perform some preprocessing on the data to improve the estimated plant results. The **Preprocess** menu gives you several options for preprocessing response data, such as removing offsets, filtering, or extracting on a subset of the data. In particular, when the response data has an offset, it is important for good identification results to remove the offset.

In the **Plant Identification** tab, click  **Preprocess** and select the preprocessing option you want to use. A tab opens with a figure that displays the

original and preprocessed data. Use the options in the tab to specify preprocessing parameters.



(For more information about preprocessing options, see “Preprocess Data” on page 6-85.)

When you are satisfied with the preprocessed signal, click **Update** to save the change to the signal. Click to return to the **Plant Identification** tab.

PID Tuner automatically adjusts the plant parameters to create a new initial guess for the plant based on the preprocessed response signal.

You can now adjust the structure and parameters of the estimated plant to obtain the estimated linear plant model for PID Tuning. See “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73 for more information.

See Also

More About

- “System Identification for PID Control” on page 6-81
- “Input/Output Data for Identification” on page 6-90
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73

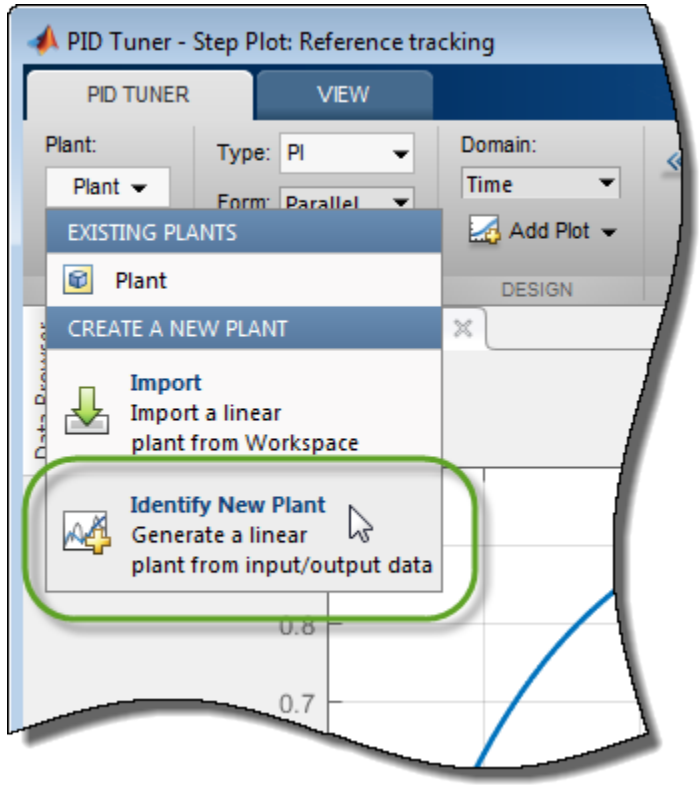
Interactively Estimate Plant from Measured or Simulated Response Data

If you have System Identification Toolbox software, **PID Tuner** lets you estimate the parameters of a linear plant model based on time-domain response data . **PID Tuner** then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink model. Plant estimation is especially useful when your Simulink model cannot be linearized or linearizes to zero.

PID Tuner gives you several techniques to graphically, manually, or automatically adjust the estimated model to match your response data. This topic illustrates some of those techniques.

Obtain Response Data for Identification

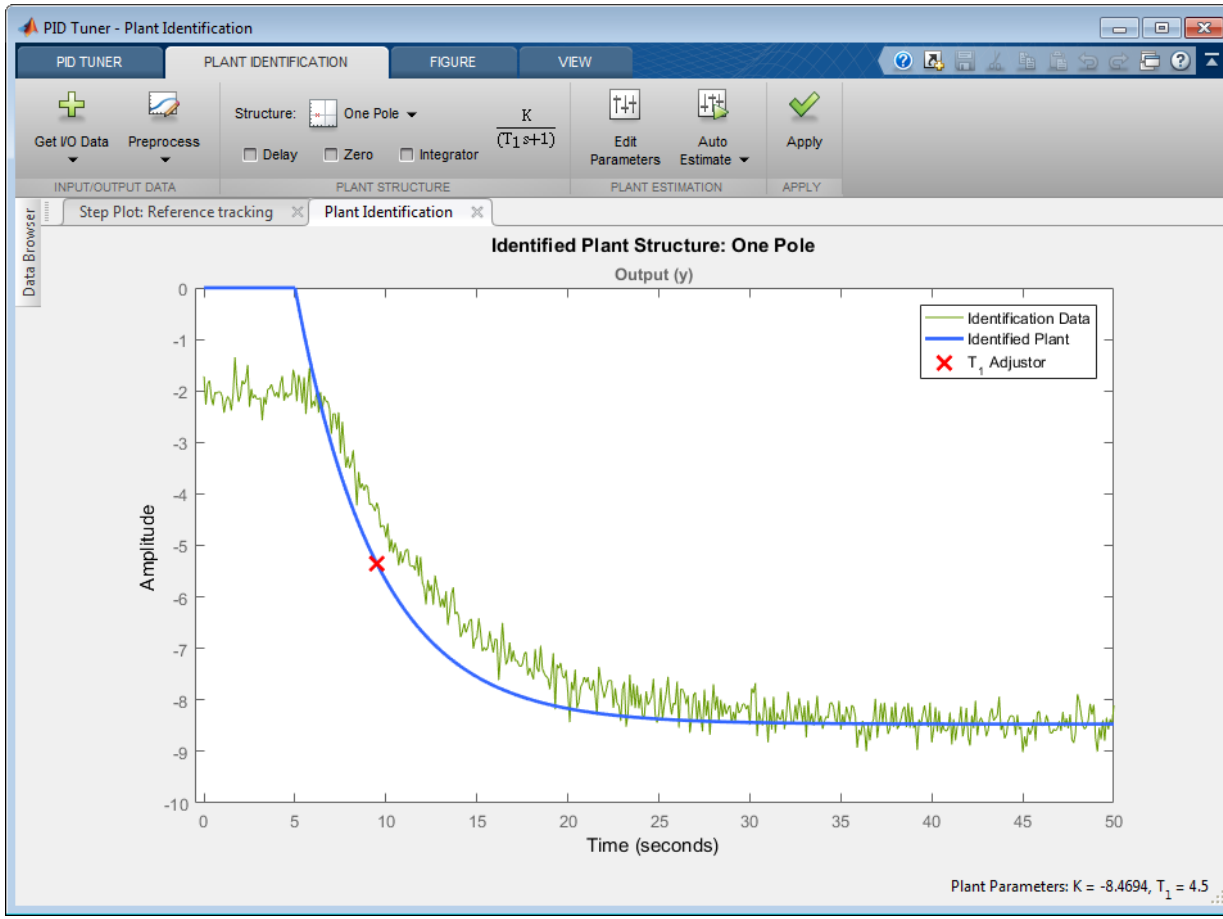
In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select `Identify New Plant`.



In the **Plant Identification** tab, click **Get I/O data**. This menu allows you to obtain system response data in one of two ways:

- **Simulate Data**. Obtain system response data by simulating the response of your Simulink model to an input signal. For more information, see Design a PID Controller Using Simulated I/O Data.
- **Import I/O Data**. Import measured system response data as described in "Import Measured Response Data for Plant Estimation" on page 6-67.

Once you have imported or simulated data, the **Plant Identification** plot displays the response data and the response of an initial estimated plant. You can now select the plant structure and adjust the estimated plant parameters until the response of the estimated plant is a good fit to the response data.



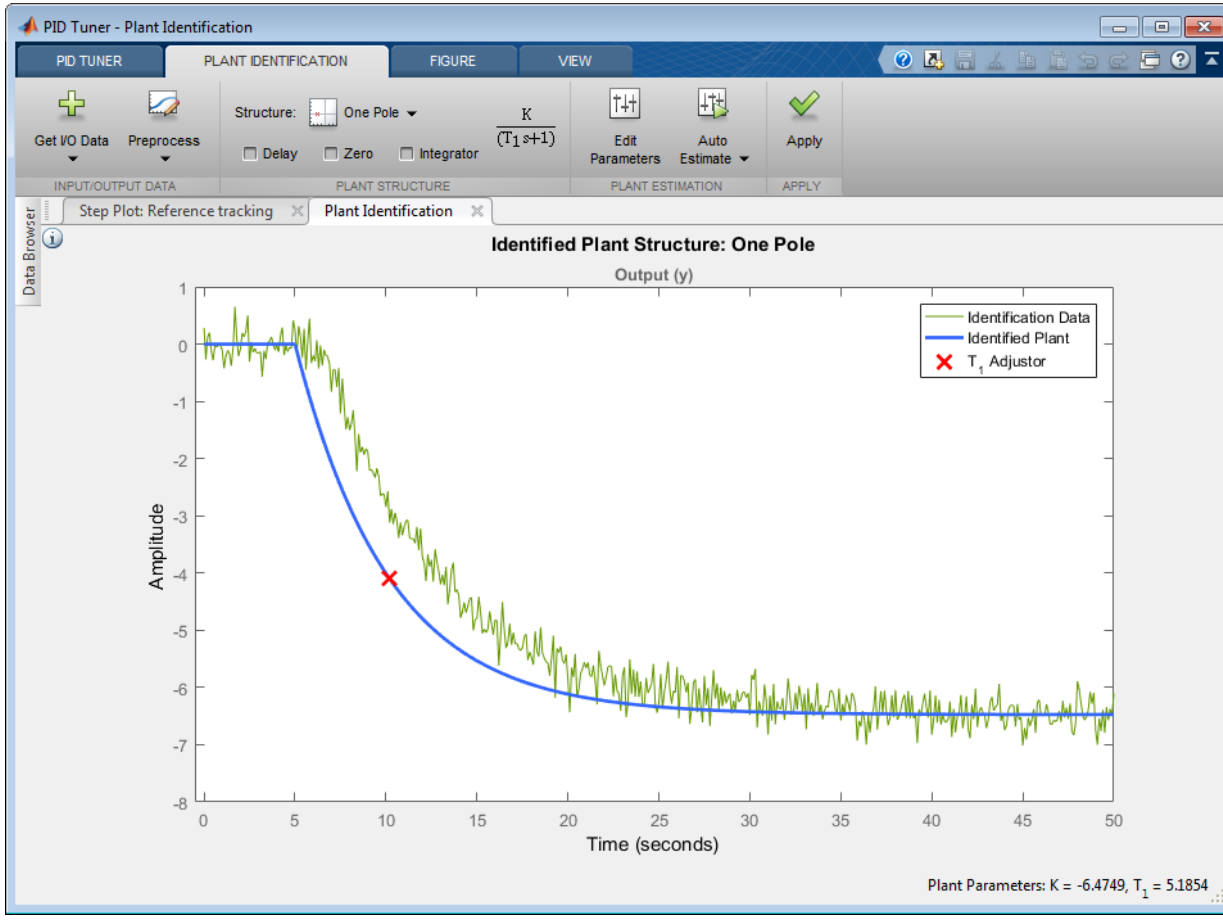
Preprocess Data

Depending on the quality and features of your imported or simulated data, you might want to perform some preprocessing on the data to improve the estimated plant results. **PID Tuner** provides several options for preprocessing response data, such as removing offsets, filtering, or extracting a subset of the data. For information, see “Preprocess Data” on page 6-85.

Adjust Plant Structure and Parameters

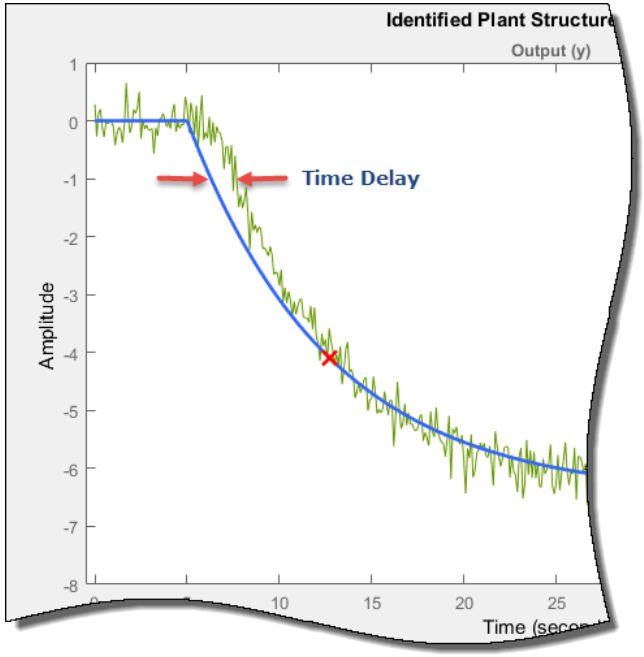
PID Tuner allows you to specify a plant structure, such as **One Pole**, **Two Real Poles**, or **State-Space Model**. In the **Structure** menu, choose the plant structure that best matches your response. You can also add a transfer delay, a zero, or an integrator to your plant.

In the following sample plot, the one-pole structure gives the qualitatively correct response. You can make further adjustments to the plant structure and parameter values to make the response of the estimated system a better match to the measured response data.



PID Tuner gives you several ways to adjust the plant parameters:

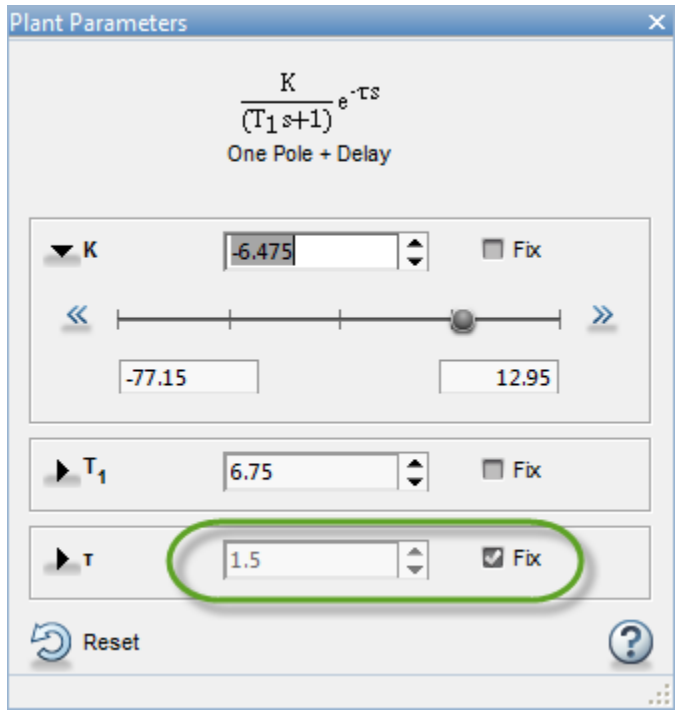
- Graphically adjust the response of the estimated system by dragging the adjustors on the plot. In this example, drag the red `x` to adjust the estimated plant time constant. **PID Tuner** recalculates system parameters as you do so. As you change the estimated system's response, it becomes apparent that there is some time delay between the application of the step input at `t` = 5 s, and the response of the system to that step input.



  To add a transport delay to the estimated plant model, in the **Plant Structure** section, check **Delay**. A vertical line appears on the plot, indicating the current value of the delay. Drag the line left or right to change the delay, and make further adjustments to the system response by dragging the red `x`.

- Adjust the numerical values of system parameters such as gains, time constants, and time delays. To numerically adjust the values of system parameters, click  **Edit Parameters**.

Suppose that you know from an independent measurement that the transport delay in your system is 1.5 seconds. In the **Plant Parameters** dialog box, enter 1.5 for τ. Check **Fix** to fix the parameter value. When you check **Fix** for a parameter, neither graphical nor automatic adjustments to the estimated plant model affect that parameter value.



- Automatically optimize the system parameters to match the measured response data.

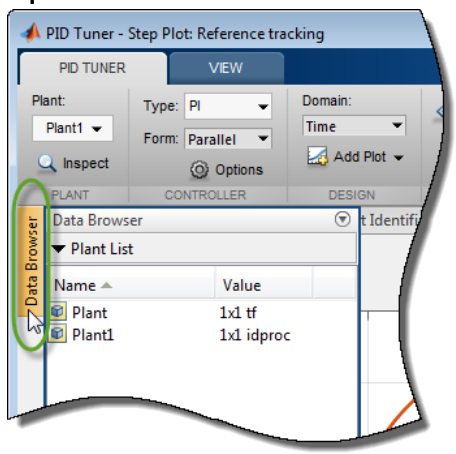
  Click **Auto Estimate** to update the estimated system parameters using the current values as an initial guess.

You can continue to iterate using any of these methods to adjust plant structure and parameter values until the estimated system's response adequately matches the measured response.

Save Plant and Tune PID Controller

When you are satisfied with the fit, click **Save Plant**. Doing so saves the estimated plant, `Plant1`, to **PID Tuner** workspace. Doing so also selects the **Step Plot: Reference Tracking** figure and returns you to the **PID Tuner** tab. **PID Tuner** automatically designs a PI controller for `Plant1`, and displays a response plot for the new closed-loop system. The **Plant** menu reflects that `Plant1` is selected for the current controller design.

Tip To examine variables stored in the **PID Tuner** workspace, open the **Data Browser**.

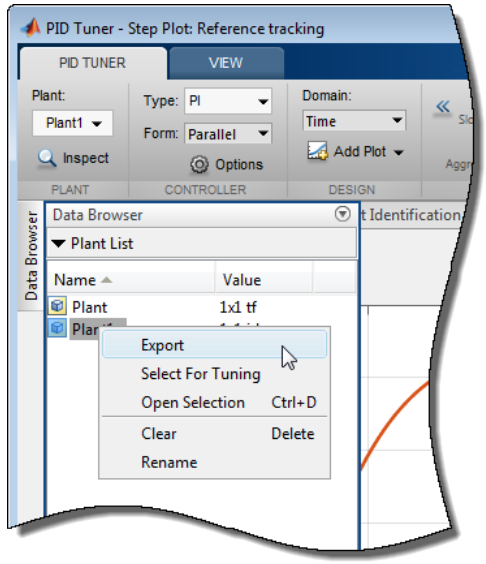


You can now use the **PID Tuner** tools to refine the controller design for the estimated plant and examine tuned system responses.

You can also export the identified plant from the **PID Tuner** workspace to the MATLAB workspace for further analysis. In the **PID Tuner** tab, click **Export**. Check the plant model you want to export to the MATLAB workspace. For this example, export `Plant1`, the plant you identified from response data. You can also export the tuned PID controller. Click **OK**. The models you selected are saved to the MATLAB workspace.

Identified plant models are saved as identified LTI models, such as `idproc` or `idss`.

Tip Alternatively, right-click a plant in the **Data Browser** to select it for tuning or export it to the MATLAB workspace.



See Also

More About

- "Choosing Identified Plant Structure" on page 6-92
- "Input/Output Data for Identification" on page 6-90
- "System Identification for PID Control" on page 6-81
- "Import Measured Response Data for Plant Estimation" on page 6-67

System Identification for PID Control

In this section...

Plant Identification

In many situations, a dynamic representation of the system you want to control is not readily available. One solution to this problem is to obtain a dynamical model using identification techniques. The system is excited by a measurable signal and the corresponding response of the system is collected at some sample rate. The resulting input-output data is then used to obtain a model of the system such as a transfer function or a state-space model. This process is called system identification or estimation. The goal of system identification is to choose a model that yields the best possible fit between the measured system response to a particular input and the model’s response to the same input.

If you have a Simulink model of your control system, you can simulate input/output data instead of measuring it. The process of estimation is the same. The system response to some known excitation is simulated, and a dynamical model is estimated based upon the resulting simulated input/output data.

Whether you use measured or simulated date for estimation, once a suitable plant model is identified, you impose control objectives on the plant based on your knowledge of the desired behavior of the system that the plant model represents. You then design a feedback controller to meet those objectives.

If you have System Identification Toolbox software, you can use **PID Tuner** for both plant identification and controller design in a single interface. You can import input/output data and use it to identify one or more plant models. Or, you can obtain simulated input/output data from a Simulink model and use that to identify one or more plant models. You can then design and verify PID controllers using these plants. **PID Tuner** also allows you to directly import plant models, such as one you have obtained from an independent identification task.

For an overview of system identification, see About System Identification (System Identification Toolbox).

Linear Approximation of Nonlinear Systems for PID Control

The dynamical behavior of many systems can be described adequately by a linear relationship between the system's input and output. Even when behavior becomes nonlinear in some operating regimes, there are often regimes in which the system dynamics are linear. For example, the behavior of an operational amplifier or the lift-vs-force dynamics of aerodynamic bodies can be described by linear models, within a certain limited operating range of inputs. For such a system, you can perform an experiment (or a simulation) that excites the system only in its linear range of behavior and collect the input/output data. You can then use the data to estimate a linear plant model, and design a PID controller for the linear model.

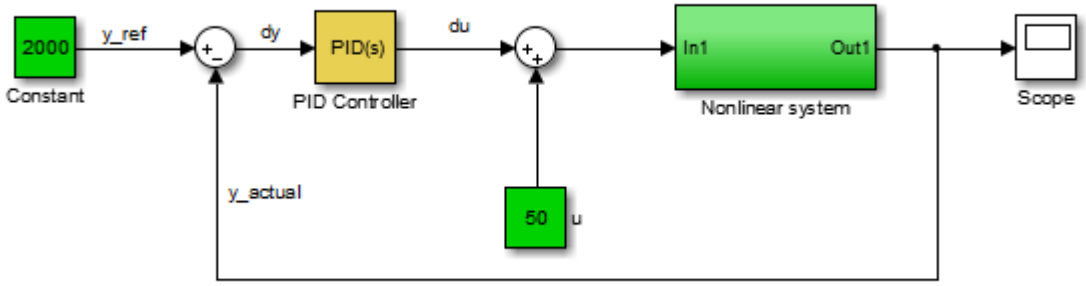
In other cases, the effects of nonlinearities are small. In such a case, a linear model can provide a good approximation, such that the nonlinear deviations are treated as disturbances. Such approximations depend heavily on the input profile, the amplitude and frequency content of the excitation signal.

Linear models often describe the deviation of the response of a system from some equilibrium point, due to small perturbing inputs. Consider a nonlinear system whose output, $y(t)$, follows a prescribed trajectory in response to a known input, $u(t)$. The dynamics are described by $dx(t)/dt = f(x, u)$, $y = g(x,u)$. Here, x is a vector of internal states of the system, and y is the vector of output variables. The functions f and g, which can be nonlinear, are the mathematical descriptions of the system and measurement dynamics. Suppose that when the system is at an equilibrium condition, a small perturbation to the input, Δu, leads to a small perturbation in the output, Δy:

$$\Delta \dot{x} = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial u}\Delta u,$$

$$\Delta y = \frac{\partial g}{\partial x}\Delta x + \frac{\partial g}{\partial u}\Delta u.$$

For example, consider the system of the following Simulink block diagram:



When operating in a disturbance-free environment, the nominal input of value 50 keeps the plant along its constant trajectory of value 2000. Any disturbances would cause the plant to deviate from this value. The PID Controller's task is to add a small correction to the input signal that brings the system back to its nominal value in a reasonable amount of time. The PID Controller thus needs to work only on the linear deviation dynamics even though the actual plant itself might be nonlinear. Thus, you might be able to achieve effective control over a nonlinear system in some regimes by designing a PID controller for a linear approximation of the system at equilibrium conditions.

Linear Process Models

A common use case is designing PID controllers for the steady-state operation of manufacturing plants. In these plants, a model relating the effect of a measurable input variable on an output quantity is often required in the form of a SISO plant. The overall system may be MIMO in nature, but the experimentation or simulation is carried out in a way that makes it possible to measure the incremental effect of one input variable on a selected output. The data can be quite noisy, but since the expectation is to control only the dominant dynamics, a low-order plant model often suffices. Such a proxy is obtained by collecting or simulating input-output data and deriving a process model (low order transfer function with unknown delay) from it. The excitation signal for deriving the data can often be a simple bump in the value of the selected input variable.

Advanced System Identification Tasks

In **PID Tuner**, you can only identify single-input, single output, continuous-time plant models. Additionally, **PID Tuner** cannot perform the following system identification tasks:

- Identify transfer functions of arbitrary number of poles and zeros. (**PID Tuner** can identify transfer functions up to three poles and one zero, plus an integrator and a time delay. **PID Tuner** can identify state-space models of arbitrary order.)
- Estimate the disturbance component of a model, which can be useful for separating measured dynamics from noise dynamics.
- Validate estimation by comparing the plant response against an independent dataset.
- Perform residual analysis.

If you need these enhanced identification features, import your data into the **System Identification** app (**System Identification**). Use the **System Identification** app to perform model identification and export the identified model to the MATLAB workspace. Then import the identified model into **PID Tuner** for PID controller design.

For more information about the System Identification Tool, see "Identify Linear Models Using System Identification App" (System Identification Toolbox).

See Also

System Identification

More About

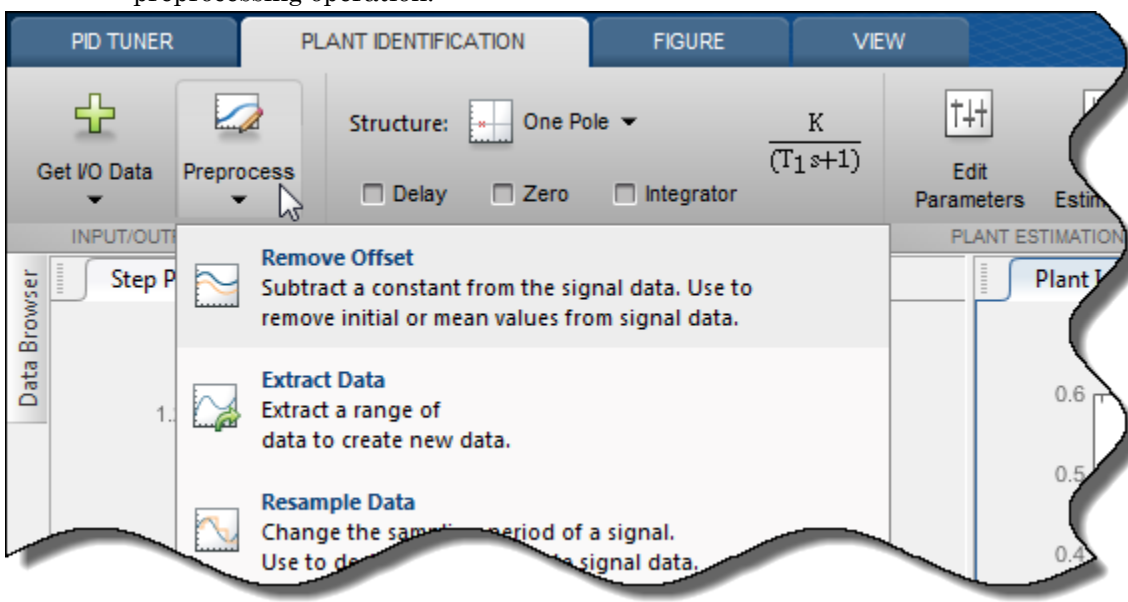
- "Input/Output Data for Identification" on page 6-90
- "Choosing Identified Plant Structure" on page 6-92
- "Interactively Estimate Plant from Measured or Simulated Response Data" on page 6-73

Preprocess Data

In this section...

Ways to Preprocess Data

In **PID Tuner**, you can preprocess plant data before you use it for estimation. After you import I/O data, on the **Plant Identification** tab, use the **Preprocess** menu to select a preprocessing operation.



- "Remove Offset" on page 6-86 — Remove mean values, a constant value, or an initial value from the data.
- "Scale Data" on page 6-86 — Scale data by a constant value, signal maximum value, or signal initial value.
- "Extract Data" on page 6-87 — Select a subset of the data to use in the . You can graphically select the data to extract, or enter start and end times in the text boxes.
- "Filter Data" on page 6-87 — Process data using a low-pass, high-pass, or band-pass filter.
- "Resample Data" on page 6-88 — Resample data using zero-order hold or linear interpolation.
- "Replace Data" on page 6-88 — 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

It is important for good results to remove data offsets. In the **Remove Offset** tab, you can remove offset from all signals at once or select a particular signal using the **Remove offset from signal** drop down list. Specify the value to remove using 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  .

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  .

Extract Data

Select a subset of data to use 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.

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 **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. Where, a normalized frequency of `1` corresponds to half the sampling rate. 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  .

Resample Data

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  .

Replace Data

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  .

Replace Data can be useful, for example, to replace outliers. Outliers can be defined as 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.

See Also

More About

- “Input/Output Data for Identification” on page 6-90
- “System Identification for PID Control” on page 6-81
- “Import Measured Response Data for Plant Estimation” on page 6-67

Input/Output Data for Identification

In this section...

Data Preparation

Identification of a plant model for PID tuning requires a single-input, single-output data set.

If you have measured data, use the data import dialogs to bring in identification data. Some common sources of identification data are transient tests such as bump test and impact test. For such data, **PID Tuner** provides dedicated dialogs that require you to specify data for only the output signal while characterizing the input by its shape. For an example, see “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73.

If you want to obtain input/output data by simulating a Simulink model, the **PID Tuner** interface lets you specify the shape of the input stimulus used to generate the response. For an example, see the Simulink Control Design example “Design a PID Controller Using Simulated I/O Data.”

Data Preprocessing

PID Tuner lets you preprocess your imported or simulated data. **PID Tuner** provides various options for detrending, scaling, and filtering the data.

It is strongly recommended to remove any equilibrium-related signal offsets from the input and output signals before proceeding with estimation. You can also filter the data to focus the signal contents to the frequency band of interest.

Some data processing actions can alter the nature of the data, which can result in transient data (step, impulse or wide pulse responses) to be treated as arbitrary input/output data. When that happens the identification plot does not show markers for adjusting the model time constants and damping coefficient.

For an example that includes a data-preprocessing step, see: “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73.

For further information about data-preprocessing options, see “Preprocess Data” on page 6-85.

Choosing Identified Plant Structure

PID Tuner provides two types of model structures for representing the plant dynamics: process models and state-space models.

Use your knowledge of system characteristics and the level of accuracy required by your application to pick a model structure. In absence of any prior information, you can gain some insight into the order of dynamics and delays by analyzing the experimentally obtained step response and frequency response of the system. For more information see the following in the System Identification Toolbox documentation:

- "Correlation Models" (System Identification Toolbox)
- "Frequency-Response Models" (System Identification Toolbox)

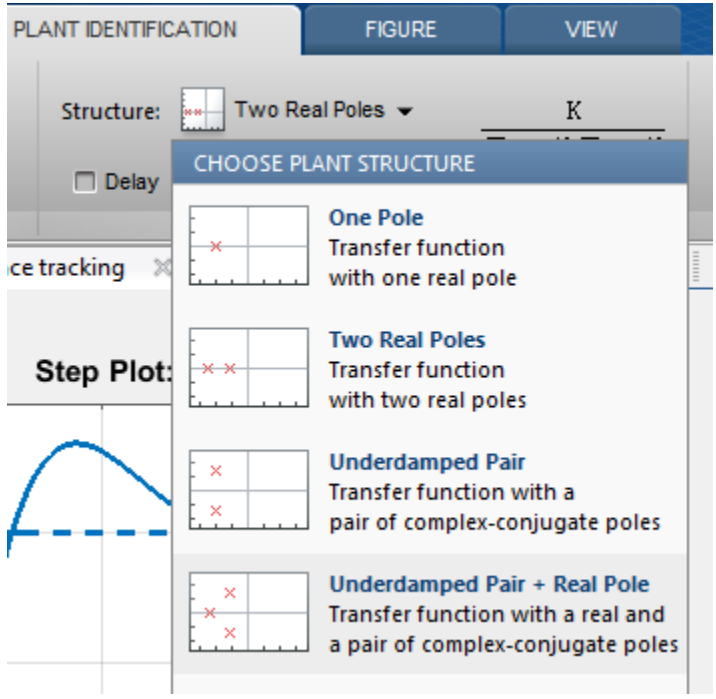
Each model structure you choose has associated dynamic elements, or model parameters. You adjust the values of these parameters manually or automatically to find an identified model that yields a satisfactory match to your measured or simulated response data. In many cases, when you are unsure of the best structure to use, it helps to start with the simplest model structure, transfer function with one pole. You can progressively try identification with higher-order structures until a satisfactory match between the plant response and measured output is achieved. The state-space model structure allows an automatic search for optimal model order based on an analysis of the input-output data.

When you begin the plant identification task, a transfer function model structure with one real pole is selected by default. This default set up is not sensitive to the nature of the data and may not be a good fit for your application. It is therefore strongly recommended that you choose a suitable model structure before performing parameter identification.

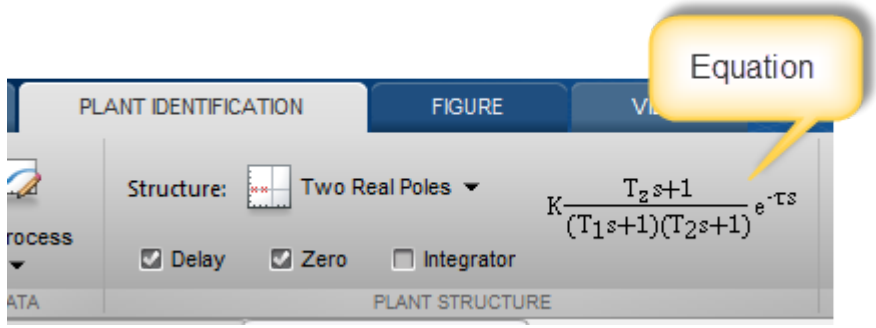
In this section...

Process Models

Process models are transfer functions with 3 or fewer poles, and can be augmented by addition of zero, delay and integrator elements. Process models are parameterized by model parameters representing time constants, gain, and time delay. In **PID Tuner**, choose a process model in the **Plant Identification** tab using the **Structure** menu.



For any chosen structure you can optionally add a delay, a zero and/or an integrator element using the corresponding checkboxes. The model transfer function configured by these choices is displayed next to the **Structure** menu.



The simplest available process model is a transfer function with one real pole and no zero or delay elements:

$$H(s)=\frac{K}{T_1 s+1}.$$

This model is defined by the parameters K, the gain, and T_1, the first time constant. The most complex process-model structure choose has three poles, an additional integrator, a zero, and a time delay, such as the following model, which has one real pole and one complex conjugate pair of poles:

$$H(s)=K\frac{T_z s+1}{s(T_1 s+1)\left(T_\omega^2 s^2+2\zeta T_\omega s+1\right)}e^{-\tau s}.$$

In this model, the configurable parameters include the time constants associated with the poles and the zero, T_1, T_ω, and T_z. The other parameters are the damping coefficient ζ, the gain K, and the time delay τ.

When you select a process model type, **PID Tuner** automatically computes initial values for the plant parameters and displays a plot showing both the estimated model response and your measured or simulated data. You can edit the parameter values graphically using indicators on the plot, or numerically using the Plant Parameters editor. For an example illustrating this process, see "Interactively Estimate Plant Parameters from Response Data" (Control System Toolbox).

The following table summarizes the various parameters that define the available types of process models.

Parameter	Used By	Description
K — Gain	All transfer functions	Can take any real value. In the plot, drag the plant response curve (blue) up or down to adjust K.
T_1 — First time constant	Transfer function with one or more real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the red x left (towards zero) or right (towards T) to adjust T_1.
T_2— Second time constant	Transfer function with two real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the magenta x left (towards zero) or right (towards T) to adjust T_2.
T_ω — Time constant associated with the natural frequency ω_n, where $T_\omega = 1/\omega_n$	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust T_ω.

Parameter	Used By	Description
ζ — Damping coefficient	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and 1. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust ζ.
τ — Transport delay	Any transfer function	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the orange vertical bar left (towards zero) or right (towards T) to adjust τ.
T_z — Model zero	Any transfer function	Can take any value between $-T$ and T, the time span of measured or simulated data. In the plot, drag the red circle left (towards $-T$) or right (towards T) to adjust T_z.
Integrator	Any transfer function	Adds a factor of $1/s$ to the transfer function. There is no associated parameter to adjust.

State-Space Models

The state-space model structure for identification is primarily defined by the choice of number of states, the model order. Use the state-space model structure when higher order models than those supported by process model structures are required to achieve a

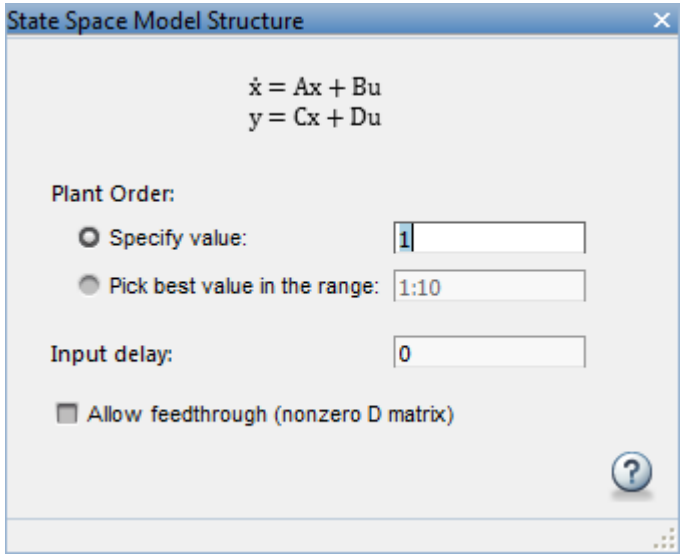
satisfactory match to your measured or simulated I/O data. In the state-space model structure, the system dynamics are represented by the state and output equations:

$$\dot{x} = Ax + Bu,$$

$$y = Cx + Du.$$

x is a vector of state variables, automatically chosen by the software based on the selected model order. u represents the input signal, and y the output signals.

To use a state-space model structure, in the **Plant Identification** tab, in the **Structure** menu, select `State-Space Model`. Then click **Configure Structure** to open the **State-Space Model Structure** dialog box.



Use the dialog box to specify model order, delay and feedthrough characteristics. If you are unsure about the order, select **Pick best value in the range**, and enter a range of orders. In this case, when you click **Estimate** in the **Plant Estimation** tab, the software displays a bar chart of Hankel singular values. Choose a model order equal to the number of Hankel singular values that make significant contributions to the system dynamics.

When you choose a state-space model structure, the identification plot shows a plant response (blue) curve only if a valid estimated model exists. For example, if you change structure after estimating a process model, the state-space equivalent of the estimated

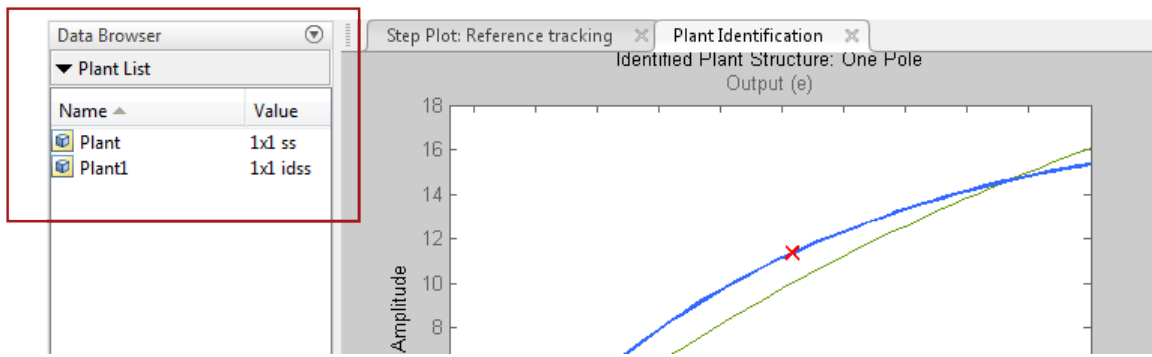
model is displayed. If you change the model order, the plant response curve disappears until a new estimation is performed.

When using the state-space model structure, you cannot directly interact with the model parameters. The identified model should thus be considered unstructured with no physical meaning attached to the state variables of the model.

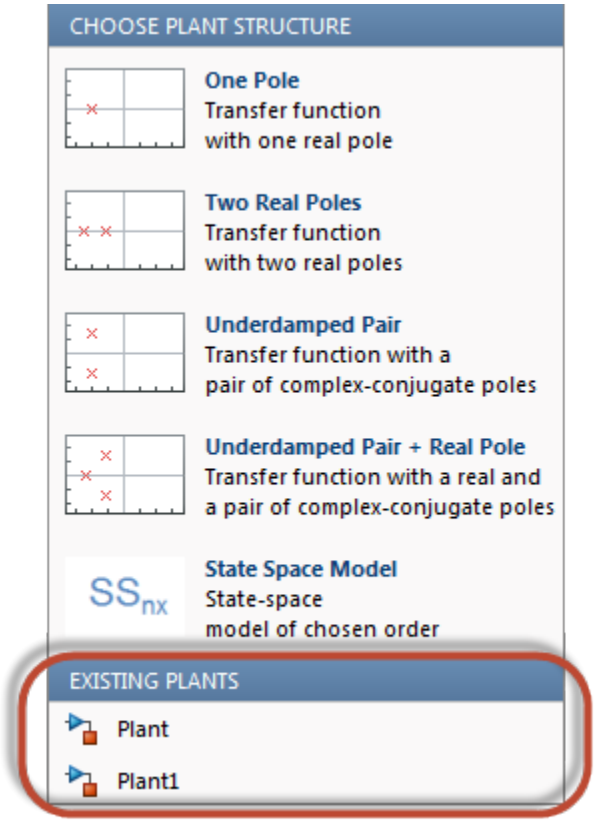
However, you can graphically adjust the input delay and the overall gain of the model. When you select a state-space model with a time delay, the delay is represented on the plot by a vertical orange bar is shown on the plot. Drag this bar horizontally to change the delay value. Drag the plant response (blue) curve up and down to adjust the model gain.

Existing Plant Models

Any previously imported or identified plant models are listed the **Plant List** section of the Data Browser.



You can define the model structure and initialize the model parameter values using one of these plants. To do so, in the **Plant Identification** tab, in the **Structure** menu, select the linear plant model you want to use for structure an initialization.



If the plant you select is a process model (`idproc` object), **PID Tuner** uses its structure. If the plant is any other model type, **PID Tuner** uses the state-space model structure.

Switching Between Model Structures

When you switch from one model structure to another, the software preserves the model characteristics (pole/zero locations, gain, delay) as much as possible. For example, when you switch from a one-pole model to a two-pole model, the existing values of T_1, T_z, τ and K are retained, T_2 is initialized to a default (or previously assigned, if any) value.

Estimating Parameter Values

Once you have selected a model structure, you have several options for manually or automatically adjusting parameter values to achieve a good match between the estimated model response and your measured or simulated input/output data. For an example that illustrates all these options, see:

- “Interactively Estimate Plant Parameters from Response Data” (Control System Toolbox) (Control System Toolbox)
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 6-73 Simulink Control Design)

PID Tuner does not perform a smart initialization of model parameters when a model structure is selected. Rather, the initial values of the model parameters, reflected in the plot, are arbitrarily-chosen middle of the range values. If you need a good starting point before manually adjusting the parameter values, use the **Initialize and Estimate** option from the **Plant Identification** tab.

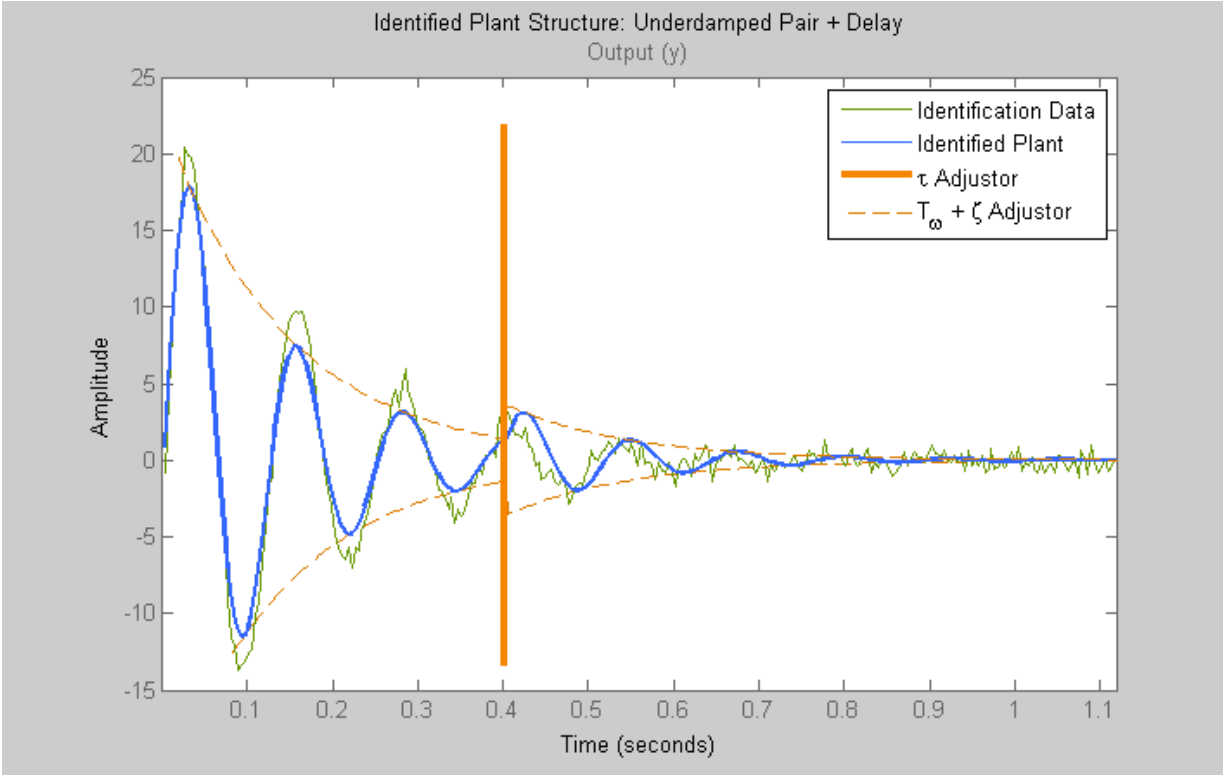
Handling Initial Conditions

In some cases, the system response is strongly influenced by the initial conditions. Thus a description of the input to output relationship in the form of a transfer function is insufficient to fit the observed data. This is especially true of systems containing weakly damped modes. **PID Tuner** allows you to estimate initial conditions in addition to the model parameters such that the sum of the initial condition response and the input response matches the observed output well. Use the **Estimation Options** dialog box to specify how the initial conditions should be handled during automatic estimation. By default, the initial condition handling (whether to fix to zero values or to estimate) is automatically performed by the estimation algorithm. However, you can enforce a certain choice by using the Initial Conditions menu.

Initial conditions can only be estimated with automatic estimation. Unlike the model parameters, they cannot be modified manually. However, once estimated they remain fixed to their estimated values, unless the model structure is changed or new identification data is imported.

If you modify the model parameters after having performed an automatic estimation, the model response will show a fixed contribution (i.e., independent of model parameters) from initial conditions. In the following plot, the effects of initial conditions were identified to be particularly significant. When the delay is adjusted afterwards, the

portion of the response to the left of the input delay marker (the τ Adjustor) comes purely from initial conditions. The portion to the right of the τ Adjustor contains the effects of both the input signal as well as the initial conditions.



See Also

More About

- "System Identification for PID Control" on page 6-81
- "Interactively Estimate Plant from Measured or Simulated Response Data" on page 6-73

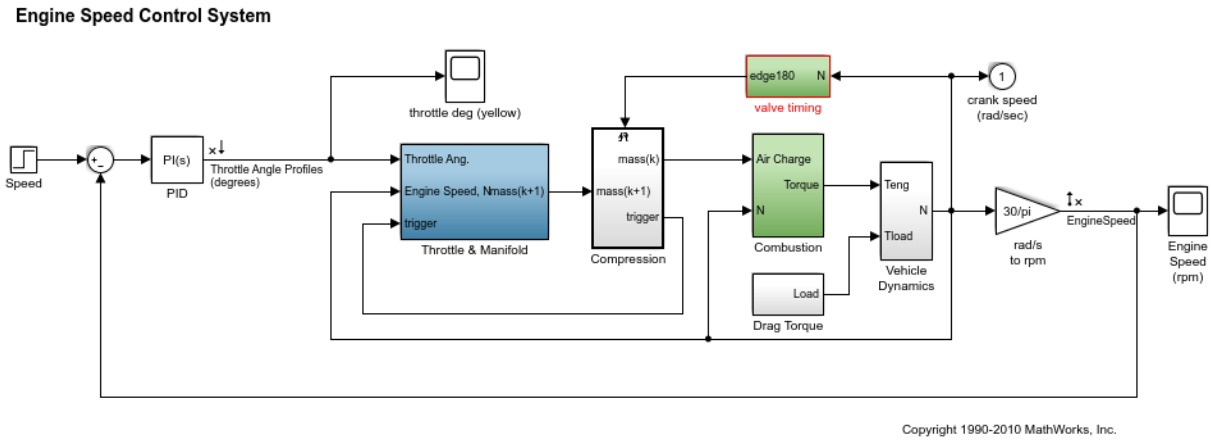
Design PID Controller Using FRD Model Obtained From "frestimate" Command

This example shows how to design a PI controller with frequency response estimated from a plant built in Simulink. This is an alternative PID design workflow when the linearized plant model is invalid for PID design (for example, when the plant model has zero gain).

Opening the Model

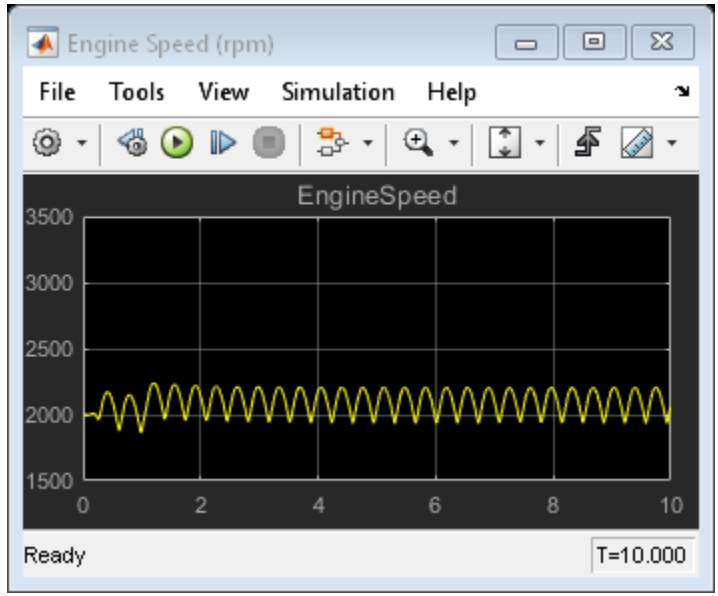
Open the engine control model and take a few moments to explore it.

```
mdl = 'scdenginectrlpidblock';
open_system(mdl)
```



The PID loop includes a PI controller in parallel form that manipulates the throttle angle to control the engine speed. The PI controller has default gains that makes the closed loop system oscillate. We want to design the controller using the PID Tuner that is launched from the PID block dialog.

```
open_system([mdl '/Engine Speed (rpm)'])
sim(mdl);
```



PID Tuner Obtaining a Plant Model with Zero Gain From Linearization

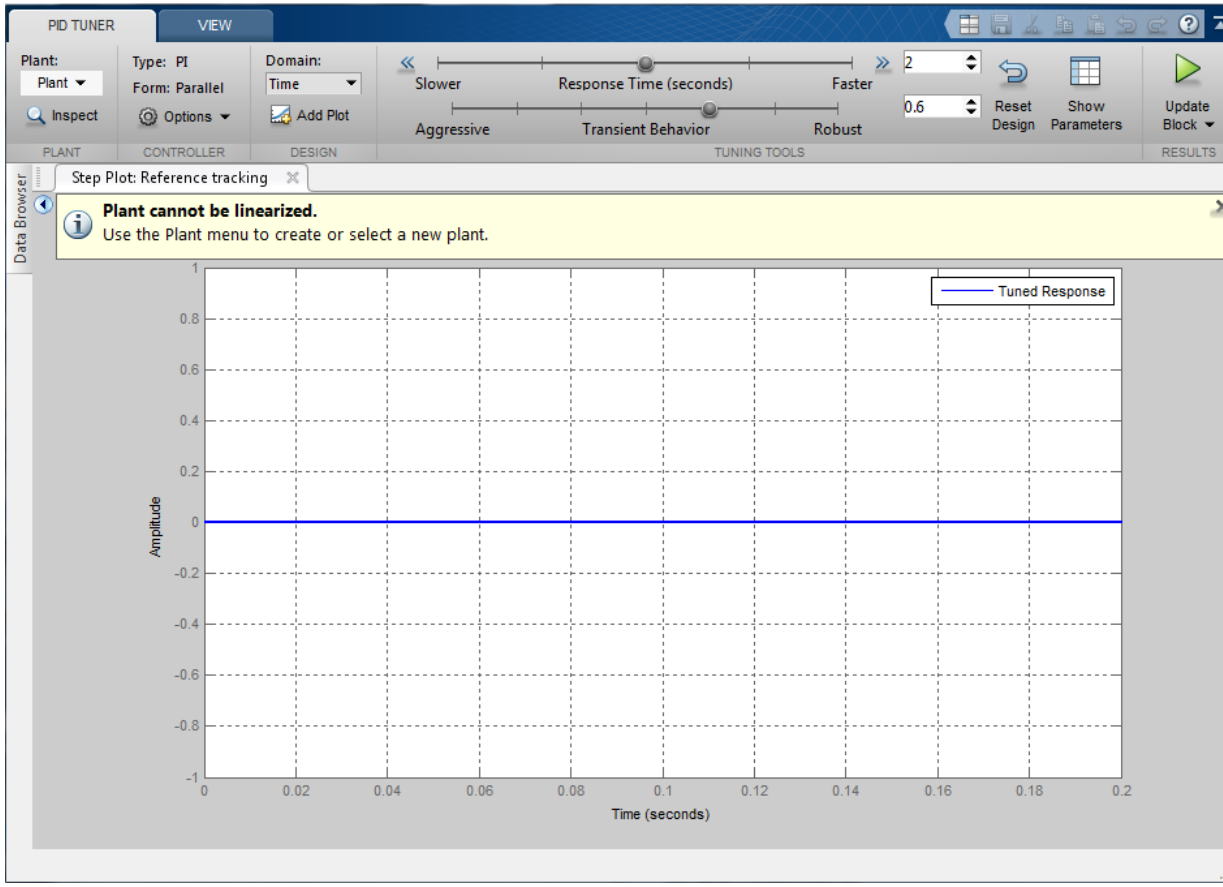
In this example, the plant seen by the PID block is from throttle angle to engine speed. Linearization input and output points are already defined at the PID block output and the engine speed measurement respectively. Linearization at the initial operating point gives a plant model with zero gain.

```
% Hide scope
close_system([mdl '/Engine Speed (rpm)'])
% Obtain the linearization input and output points
io = getlinio(mdl);
% Linearize the plant at initial operating point
linsys = linearize(mdl,io)
```

```
linsys =

  D =
                Throttle Ang
   EngineSpeed             0

Static gain.
```

The reason for obtaining zero gain is that there is a triggered subsystem "Compression" in the linearization path and the analytical block-by-block linearization does not support events-based subsystems. Since the PID Tuner uses the same approach to obtain a linear plant model, the PID Tuner also obtains a plant model with zero gain and reject it during the launching process.

To launch the PID Tuner, open the PID block dialog and click Tune button. An information dialog shows up and indicates that the plant model linearized at initial operating point has zero gain and cannot be used to design a PID controller.



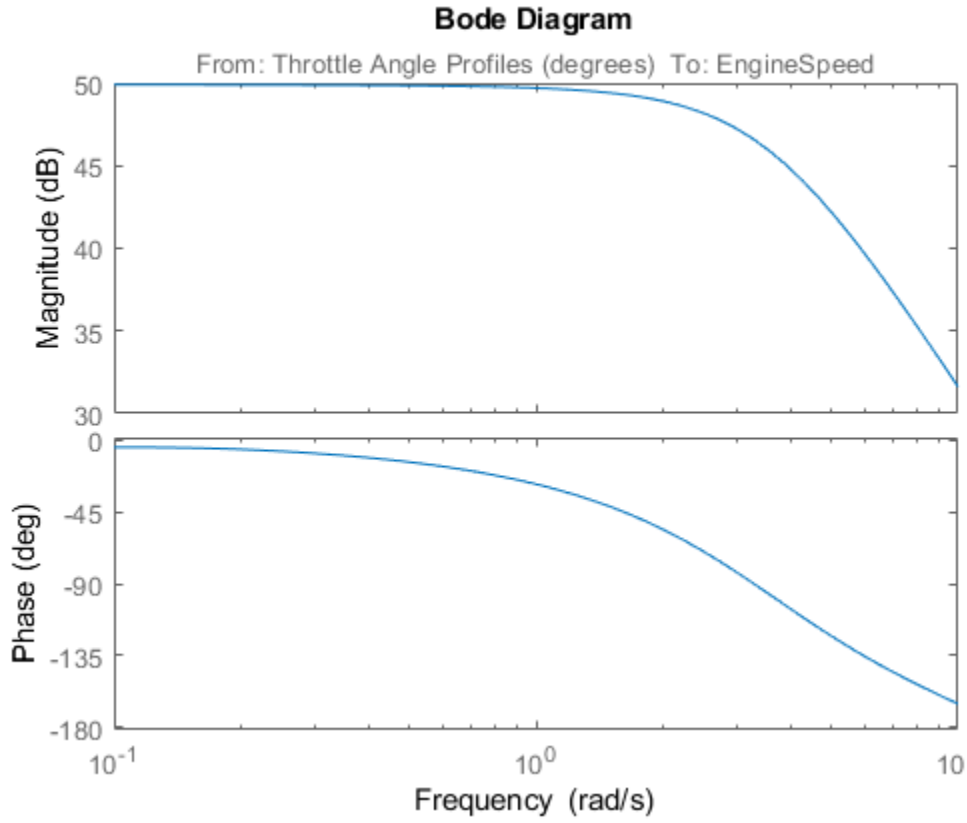
The alternative way to obtain a linear plant model is to directly estimate the frequency response data from the Simulink model, create an FRD system in MATLAB Workspace, and import it back to the PID Tuner to continue PID design.

Obtaining Estimated Frequency Response Data Using Sinestream Signals

Sinestream input signal is the most reliable input signal for estimating an accurate frequency response of a Simulink model using `frestimate` command. More information on how to use `frestimate` can be found in the example "Frequency Response Estimation Using Simulation-Based Techniques" in Simulink Control Design examples.

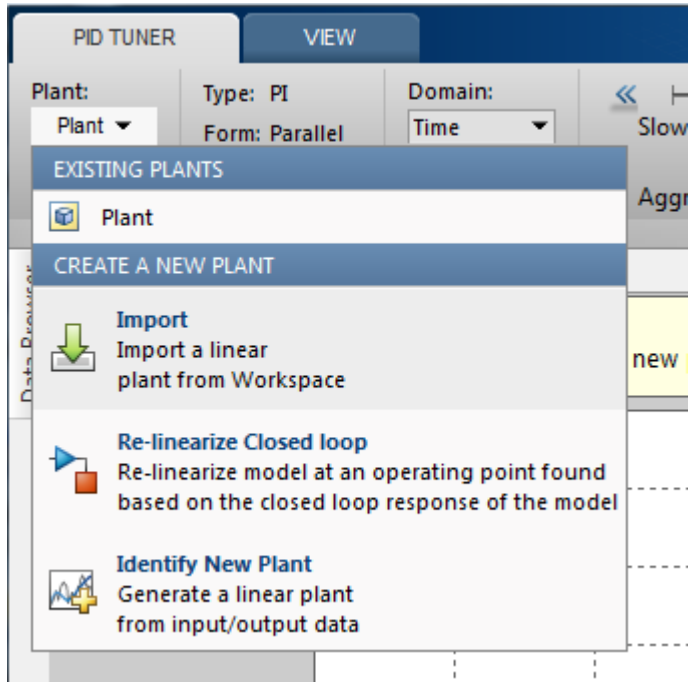
In this example, we create a sine stream that sweeps frequency from 0.1 to 10 rad/sec. Its amplitude is set to be 1e-3. You can inspect the estimation results using the bode plot.

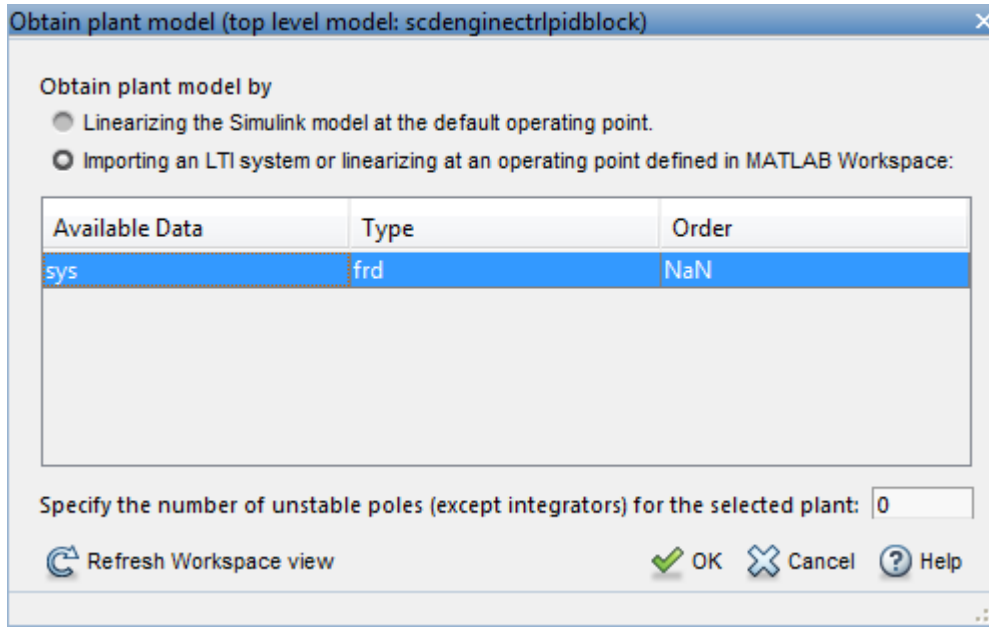
```
% Construct sine signal
in = frest.Sinestream('Frequency',logspace(-1,1,50),'Amplitude',1e-3);
% Estimate frequency response
sys = frestimate(mdl,io,in); % this command may take a few minutes to finish
% Display Bode plot
figure;
bode(sys);
```



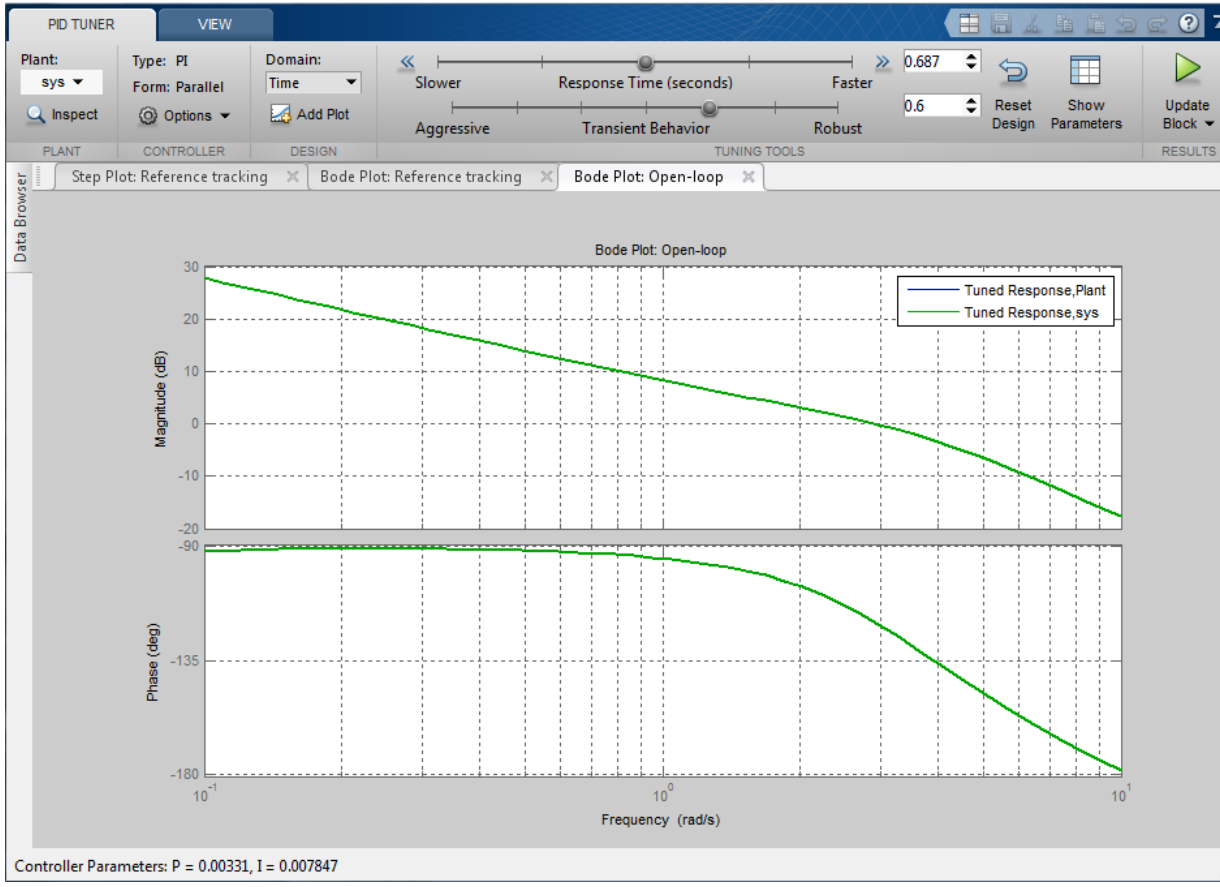
Designing PI with the FRD System in PID Tuner

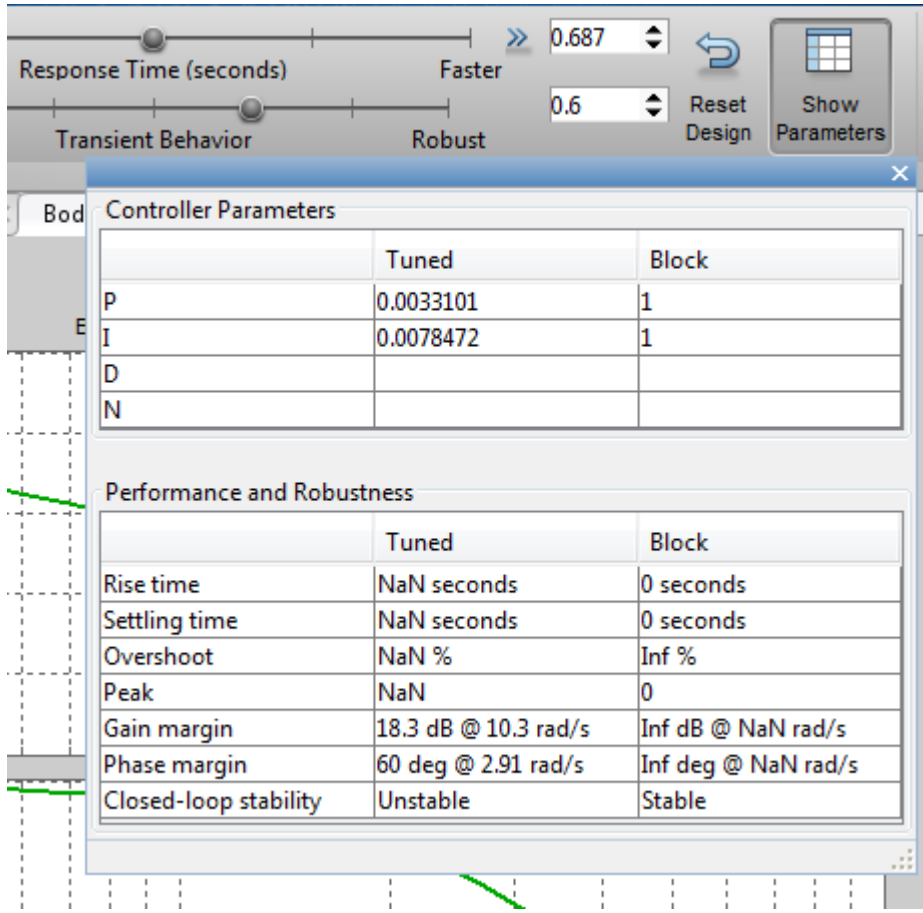
SYS is a FRD system that represents the plant frequency response at the initial operating point. To use it in the PID Tuner, we need to import it after the Tuner is launched. Click **Plant** and select **Import**.


PID TUNER
VIEW
Plant:
Plant
Type: PI
Form: Parallel
Domain:
Time
Slow
EXISTING PLANTS
Plant
CREATE A NEW PLANT
Import
Import a linear
plant from Workspace
Re-linearize Closed loop
Re-linearize model at an operating point found
based on the closed loop response of the model
Identify New Plant
Generate a linear plant
from input/output data
new



Click the 2nd radio button, select "sys" from the list, and click "OK" to import the FRD system into the PID Tuner. The automated design returns a stabilizing controller. Click **Add Plot** and select **Open-Loop** Bode plot. The plot shows reasonable gain and phase margin. Click **Show Parameters** to see the gain and phase margin values. Time domain response plots are not available for FRD plant models.

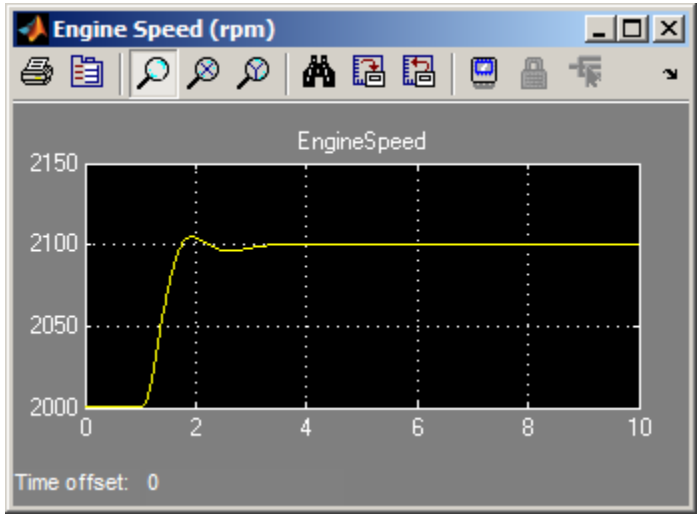

PID TUNER
VIEW
Plant:
sys
Inspect
Type: PI
Form: Parallel
Options
Domain:
Time
Add Plot
Slower
Response Time (seconds)
Faster
0.687
Aggressive
Transient Behavior
Robust
0.6
Reset Design
Show Parameters
Update Block
PLANT
CONTROLLER
DESIGN
TUNING TOOLS
RESULTS
Data Browser
Step Plot: Reference tracking
Bode Plot: Reference tracking
Bode Plot: Open-loop
Bode Plot: Open-loop
Tuned Response,Plant
Tuned Response,sys
Magnitude (dB)
Phase (deg)
Frequency (rad/s)
Controller Parameters: P = 0.00331, I = 0.007847



Click **Update Block** to update the PID block P and I gains to the PID.

Simulating Closed-Loop Performance in Simulink Model

Simulation in Simulink shows that the new PI controller provides good performance when controlling the nonlinear model.



Close the model.

```
bdclose(mdl);
```

Designing a Family of PID Controllers for Multiple Operating Points

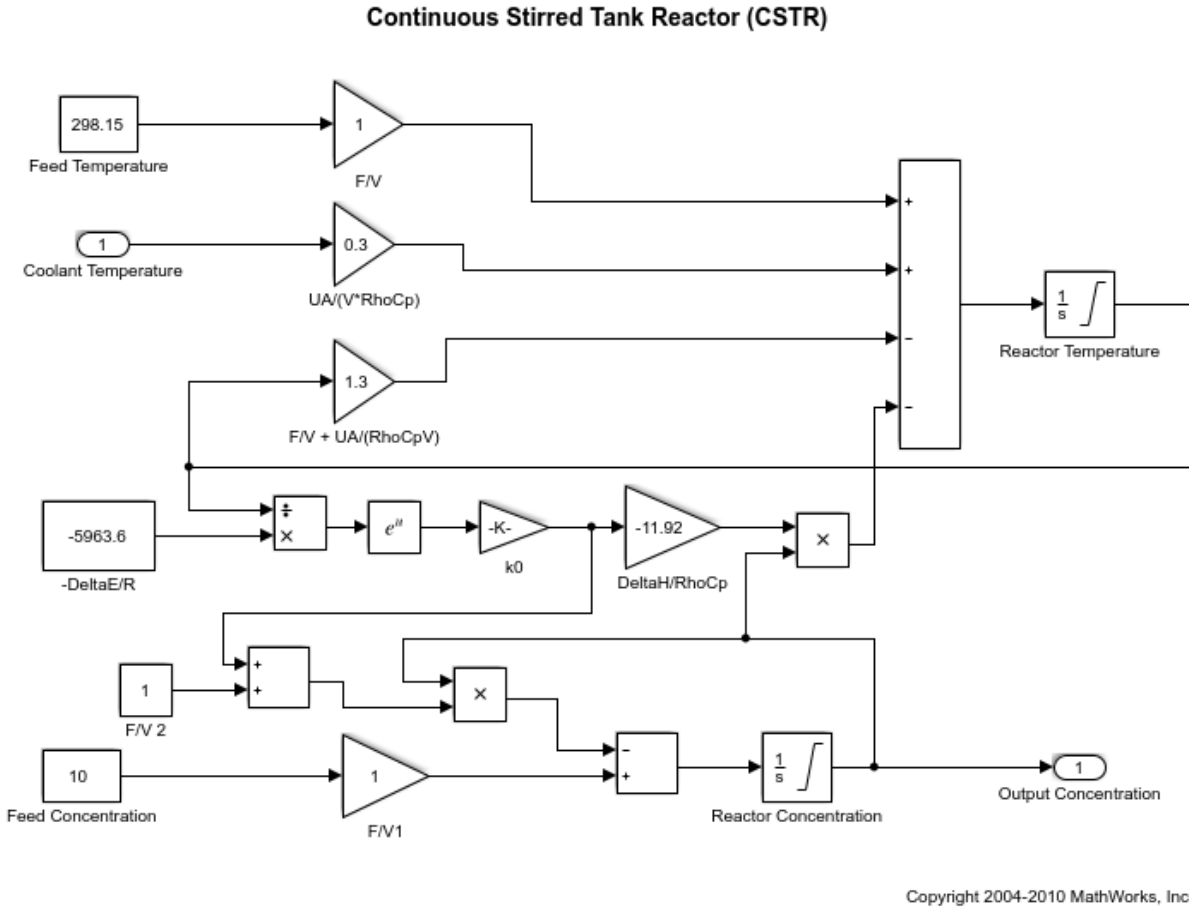
This example shows how to design an array of PID controllers for a nonlinear plant in Simulink that operates over a wide range of operating points.

Opening the Plant Model

The plant is a continuous stirred tank reactor (CSTR) that operates over a wide range of operating points. A single PID controller can effectively use the coolant temperature to regulate the output concentration around a small operating range that the PID controller is designed for. But since the plant is a strongly nonlinear system, control performance degrades if operating point changes significantly. The closed-loop system can even become unstable.

Open the CSTR plant model.

```
mdl = 'scdcstrctrlplant';
open_system(mdl)
```



For background, see Seborg, D.E. et al., "Process Dynamics and Control", 2nd Ed., 2004, Wiley, pp.34-36.

Introduction to Gain Scheduling

A common approach to solve the nonlinear control problem is using gain scheduling with linear controllers. Generally speaking designing a gain scheduling control system takes four steps:

1. Obtain a plant model for each operating region. The usual practice is to linearize the plant at several equilibrium operating points.
2. Design a family of linear controllers such as PID for the plant models obtained in the previous step.
3. Implement a scheduling mechanism such that the controller coefficients such as PID gains are changed based on the values of the scheduling variables. Smooth (bumpless) transfer between controllers is required to minimize disturbance to plant operation.
4. Assess control performance with simulation.

For more background reading on gain scheduling, see a survey paper from W. J. Rugh and J. S. Shamma: "Research on gain scheduling", Automatica, Issue 36, 2000, pp. 1401-1425.

In this example, we focus on designing a family of PID controllers for the CSTR plant described in step 1 and 2.

Obtaining Linear Plant Models for Multiple Operating Points

The output concentration C is used to identify different operating regions. The CSTR plant can operate at any conversion rate between low conversion rate (C=9) and high conversion rate (C=2). In this example, divide the whole operating range into 8 regions represented by C = 2, 3, 4, 5, 6, 7, 8 and 9.

In the following, first compute equilibrium operating points with the `findop` command. Then linearize the plant at each operating point with the `linearize` command.

```
% Specify operating regions
C = [2 3 4 5 6 7 8 9];
% Obtain default operating point array
op = operspec(mdl, numel(C));
% Initialize an array of operating point specifications
for ct = 1:numel(C)
    % Set the value of output concentration C to be known
    op(ct).Outputs.Known = true;
    % Compute equilibrium operating point corresponding to the value of C
    op(ct).Outputs.y = C(ct);
end
% Compute equilibrium operating point corresponding to the value of C
opoint = findop(mdl,op,findopOptions('DisplayReport','off'));
% Linearize plant
Plants = linearize(mdl, opoint);
```

Since the CSTR plant is nonlinear, we expect different characteristics among the linear models. For example, plant models with high and low conversion rates are stable, while the others are not.

```
isstable(Plants,'elem')'
```

```
ans =

  1x8 logical array

   1   1   0   0   0   0   1   1
```

Designing PID Controllers for the Plant Models

To design multiple PID controllers in batch, we can use the `pidtune` command. The following command will generate an array of PID controllers in parallel form. The desired open loop crossover frequency is at 1 rad/sec and the phase margin is the default value of 60 degrees.

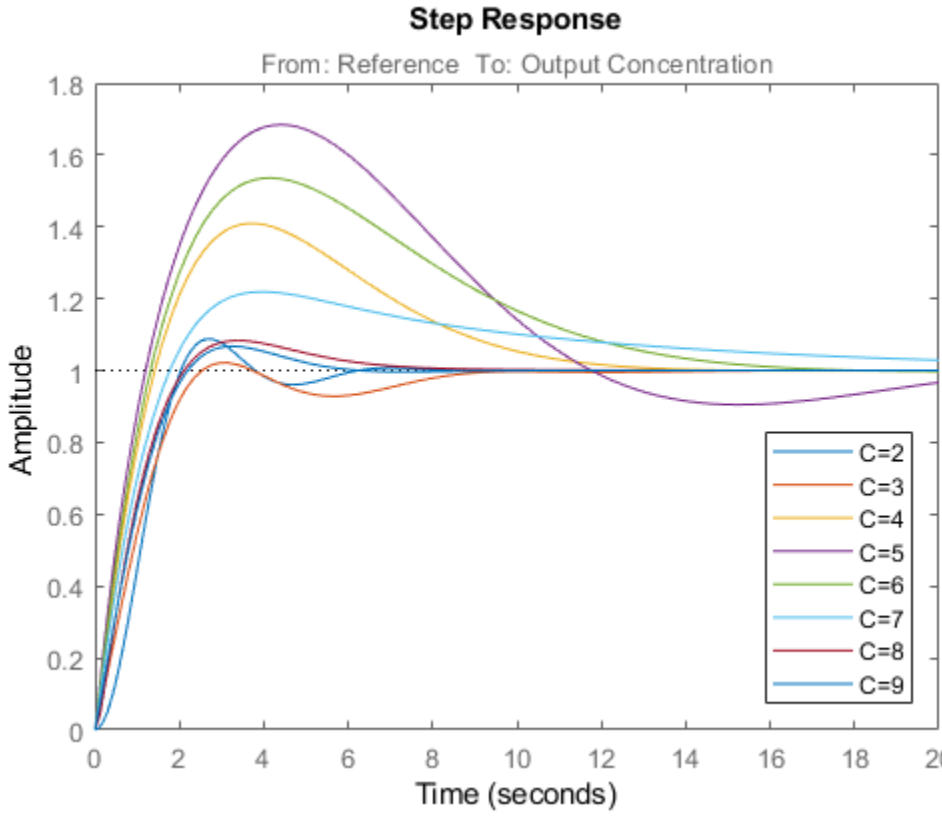
```
% Design controllers
Controllers = pidtune(Plants,'pidf',pidtuneOptions('Crossover',1));
% Display controller for C=4
Controllers(:,:,4)
```

```
ans =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = -12.4, Ki = -1.74, Kd = -16, Tf = 0.00875

Continuous-time PIDF controller in parallel form.
```

Plot the closed loop responses for step set-point tracking as below:

```
% Construct closed-loop systems
clsys = feedback(Plants*Controllers,1);
% Plot closed-loop responses
figure;
hold on
```

```
for ct = 1:length(C)
    % Select a system from the LTI array
    sys = clsys(:,:,ct);
    sys.Name = ['C=',num2str(C(ct))];
    sys.InputName = 'Reference';
    % Plot step response
    stepplot(sys,20);
end
legend('show','location','southeast')
```



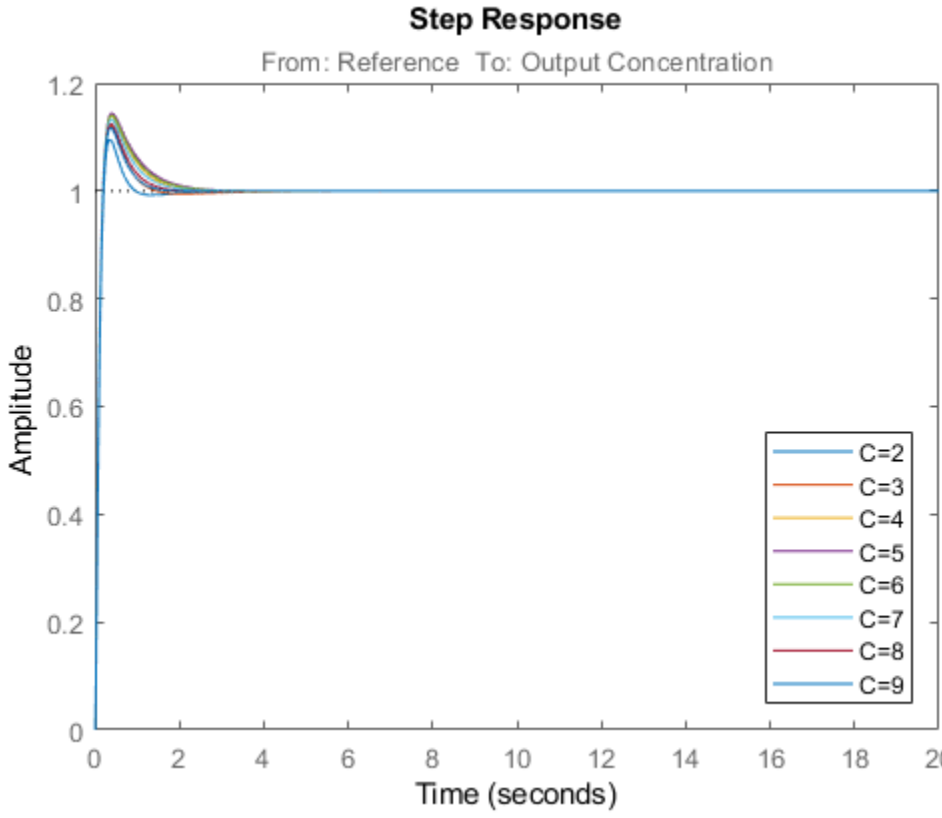
All the closed loops are stable but the overshoots of the loops with unstable plants (C=4, 5, 6, and 7) are too large. To improve the results, increase the target open loop bandwidth to 10 rad/sec.

```
% Design controllers for unstable plant models
Controllers = pidtune(Plants,'pidf',10);
% Display controller for C=4
Controllers(:,:,4)
```

```
ans =

                1             s
  Kp + Ki * --- + Kd * --------
                s          Tf*s+1

  with Kp = -283, Ki = -151, Kd = -128, Tf = 0.0183

Continuous-time PIDF controller in parallel form.
```

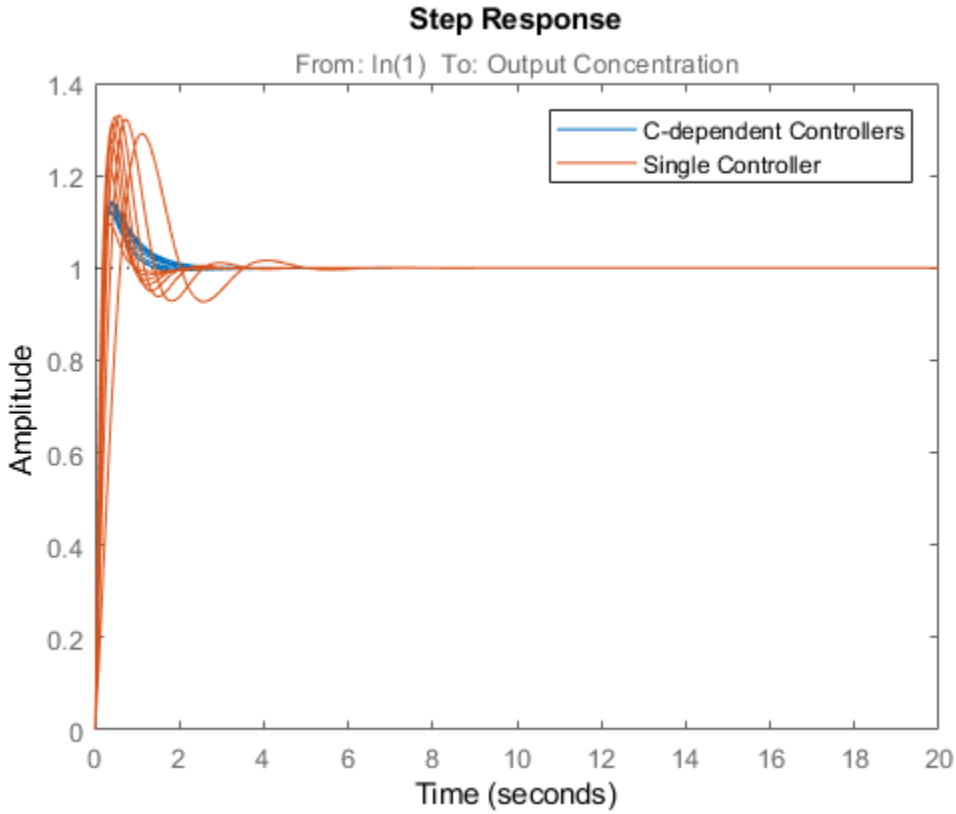
Plot the closed-loop step responses for the new controllers.

```
% Construct closed-loop systems
clsys = feedback(Plants*Controllers,1);
% Plot closed-loop responses
figure;
hold on
for ct = 1:length(C)
    % Select a system from the LTI array
    sys = clsys(:,:,ct);
    set(sys,'Name',['C=',num2str(C(ct))],'InputName','Reference');
    % Plot step response
    stepplot(sys,20);
end
legend('show','location','southeast')
```



All the closed loop responses are satisfactory now. For comparison, examine the response when you use the same controller at all operating points. Create another set of closed-loop systems, where each one uses the C = 2 controller.

```
clsys_flat = feedback(Plants*Controllers(:,:,1),1);

figure;
stepplot(clsys,clsys_flat,20)
legend('C-dependent Controllers','Single Controller')
```



The array of PID controllers designed separately for each concentration gives considerably better performance than a single controller.

However, the closed-loop responses shown above are computed based on linear approximations of the full nonlinear system. To validate the design, implement the scheduling mechanism in your model using the PID Controller block.

Close the model.

```
bdclose(mdl);
```

See Also

`findop` | `operspec` | `pidtune`

More About

- “Implement Gain-Scheduled PID Controllers” on page 6-121

Implement Gain-Scheduled PID Controllers

This example shows how to implement gain-scheduled control in a Simulink model using a family of PID controllers. The PID controllers are tuned for a series of steady-state operating points of the plant, which is highly nonlinear.

This example builds on the work done in "Designing a Family of PID Controllers for Multiple Operating Points" on page 6-112. In that example, the continuous stirred tank reactor (CSTR) plant model is linearized at steady-state operating points that have output concentrations $C = 2, 3, ..., 8, 9$. The nonlinearity in the CSTR plant yields different linearized dynamics at different output concentrations. The example uses the `pidtune` command to generate and tune a separate PID controller for each output concentration.

You can expect each controller to perform well in a small operating range around its corresponding output concentration. This example shows how to use the PID Controller block to implement all of these controllers in a gain-scheduled configuration. In such a configuration, the PID gains change as the output concentration changes. This configuration ensures good PID control at any output concentration within the operating range of the control system.

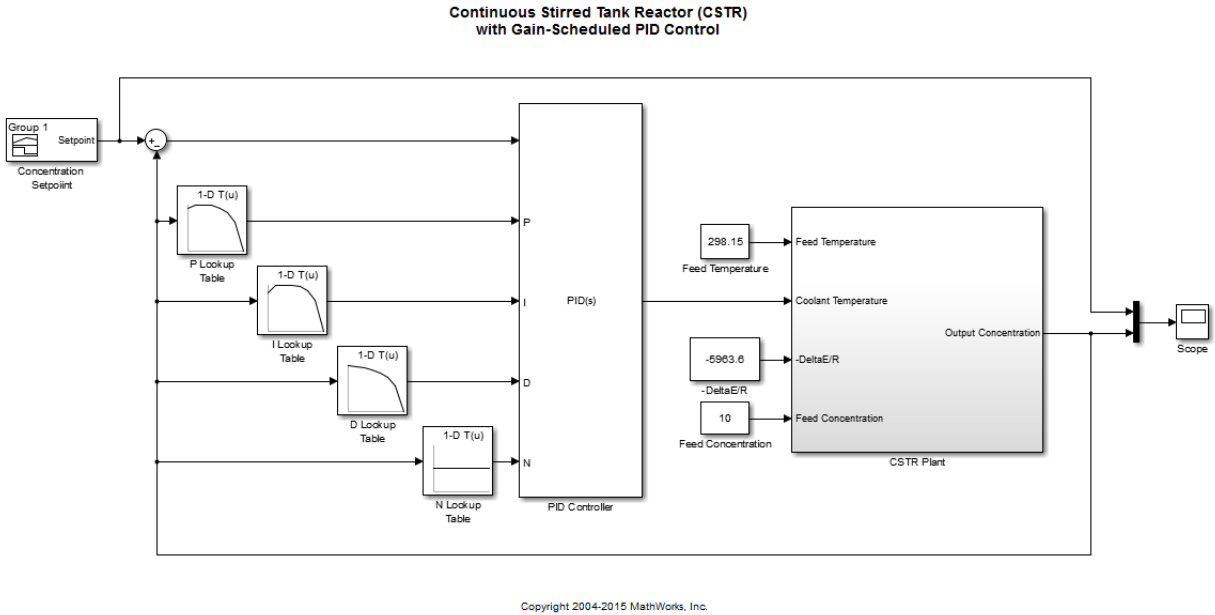
Begin with the controllers generated in "Designing a Family of PID Controllers for Multiple Operating Points" on page 6-112. If these controllers are not already in the MATLAB workspace, load them from the data file `PIDGainSchedExample.mat`.

```
load PIDGainSchedExample
```

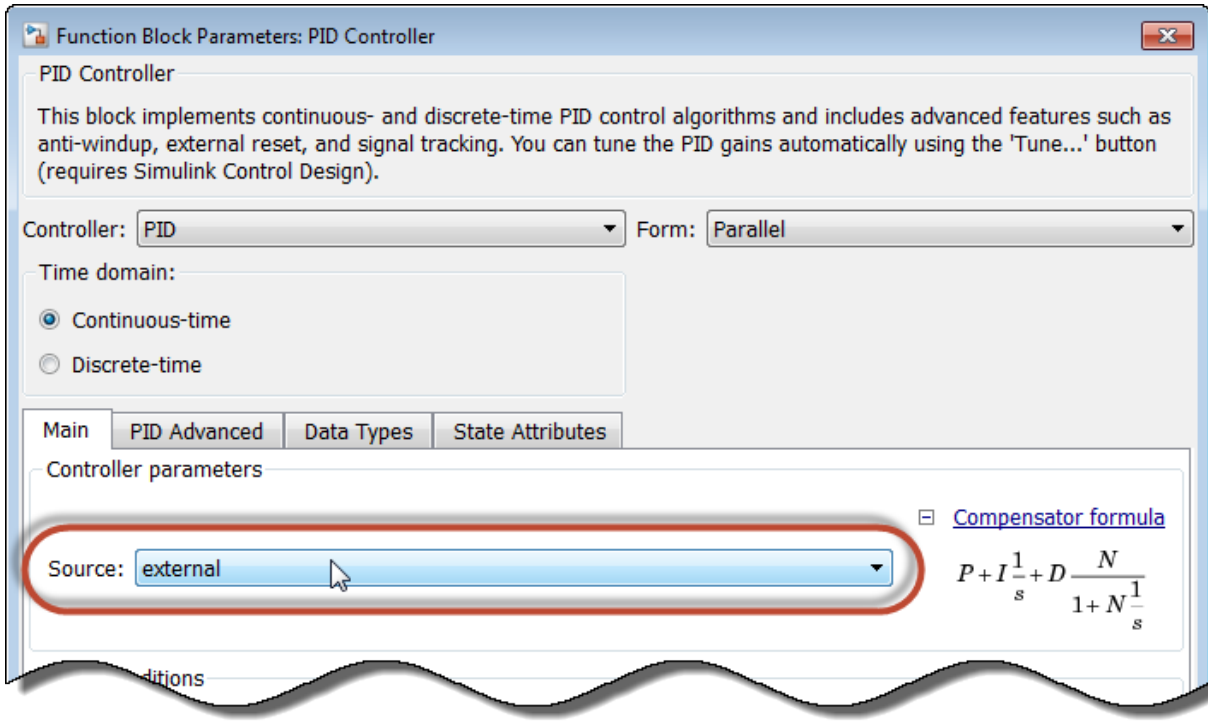
This operation puts two variables in the MATLAB workspace, `Controllers` and `C`. The model array `Controllers` contains eight `pid` models, each tuned for one output concentration in the vector `C`.

To implement these controllers in a gain-scheduled configuration, create lookup tables that associate each output concentration with the corresponding set of PID gains. The Simulink model `PIDGainSchedCSTRExampleModel` contains such lookup tables, configured to provide gain-scheduled control for the CSTR plant. Open this model.

```
open_system('PIDGainSchedCSTRExampleModel')
```

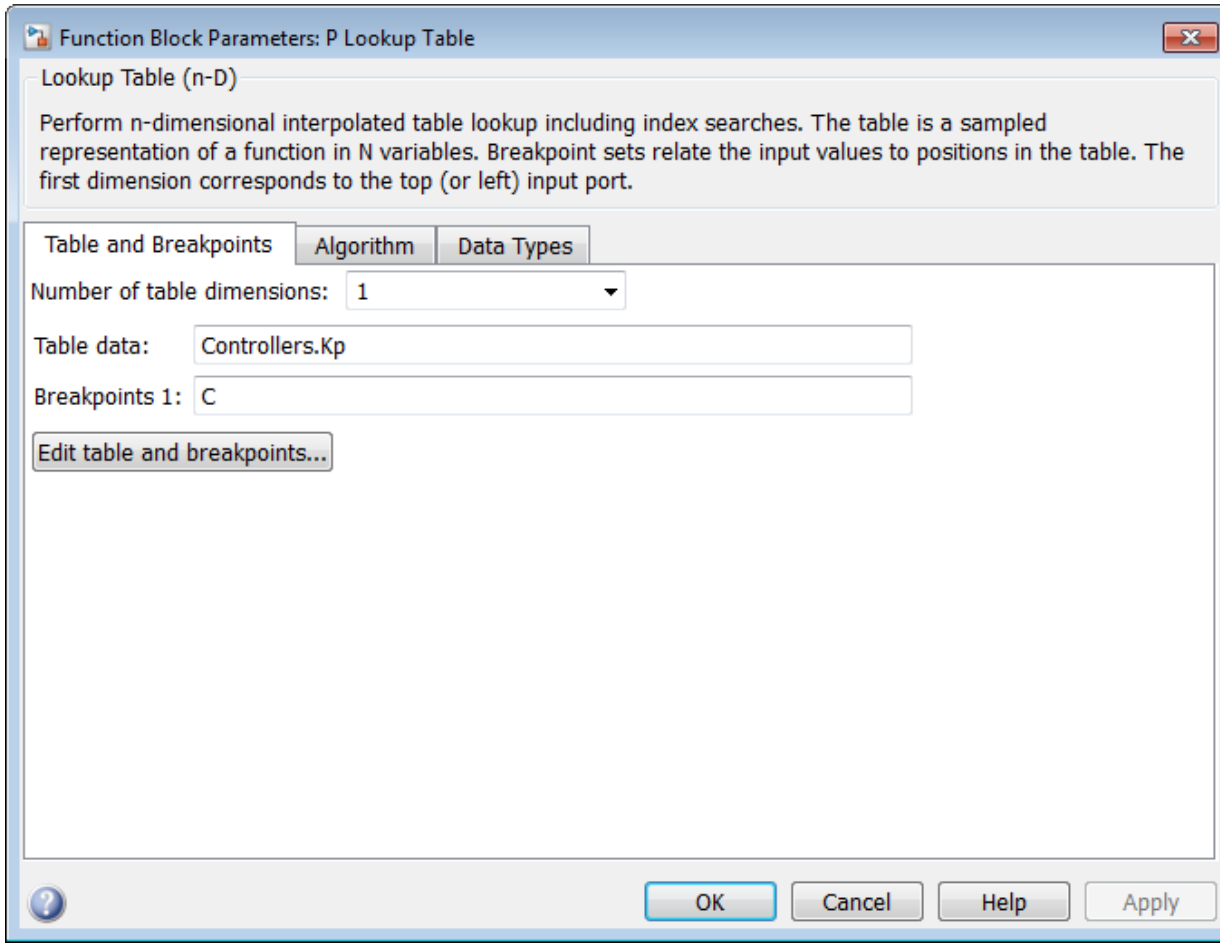


In this model, the PID Controller block is configured to have external input ports for the PID coefficients. Using external inputs allows the coefficients to vary as the output concentration varies. Double-click the block to examine the configuration.



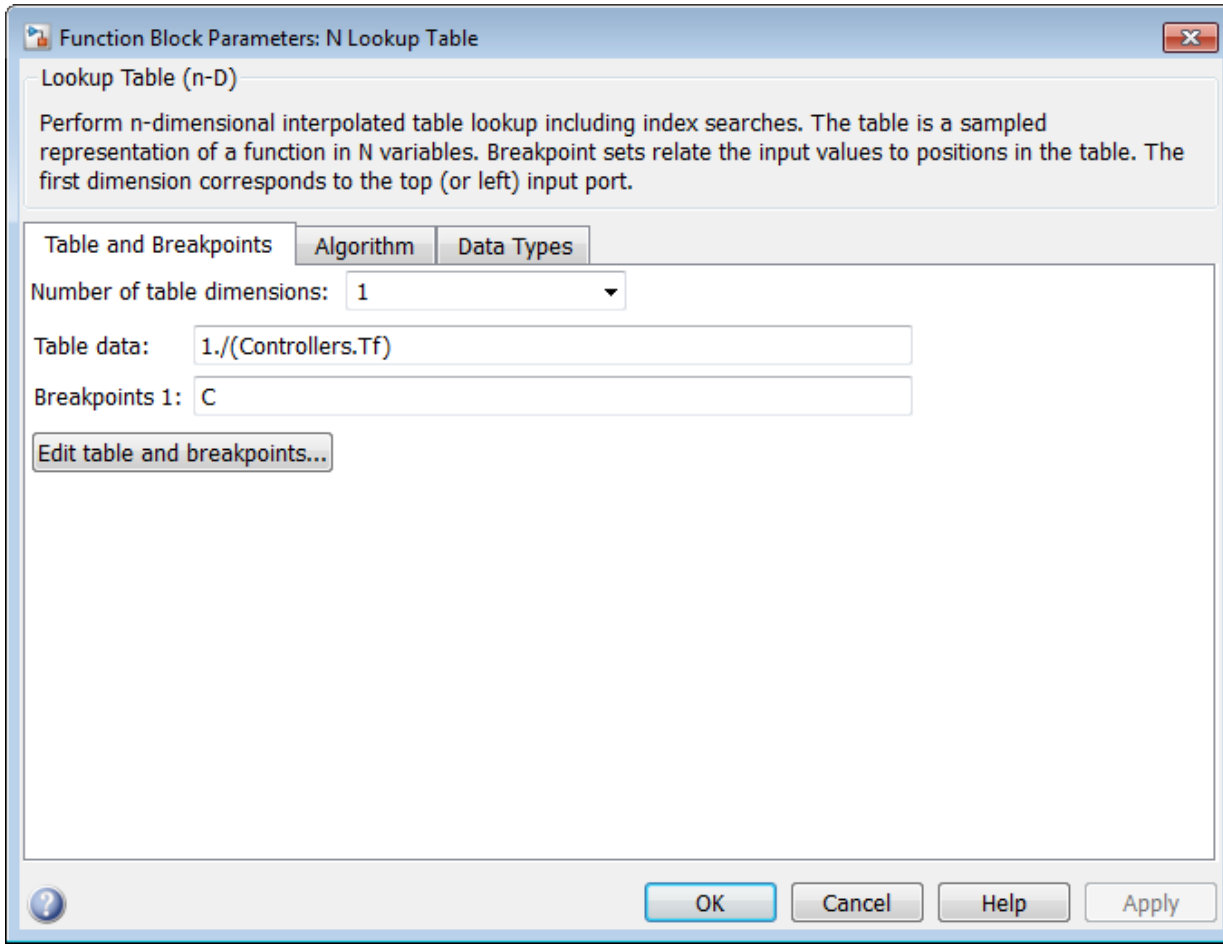
Setting the controller parameters **Source** to `external` enables the input ports for the coefficients.

The model uses a 1-D Lookup Table block for each of the PID coefficients. In general, for gain-scheduled PID control, use your scheduling variable as the lookup-table input, and the corresponding controller coefficient values as the output. In this example, the CSTR plant output concentration is the lookup table input, and the output is the PID coefficient corresponding to that concentration. To see how the lookup tables are configured, double-click the `P Lookup Table` block.



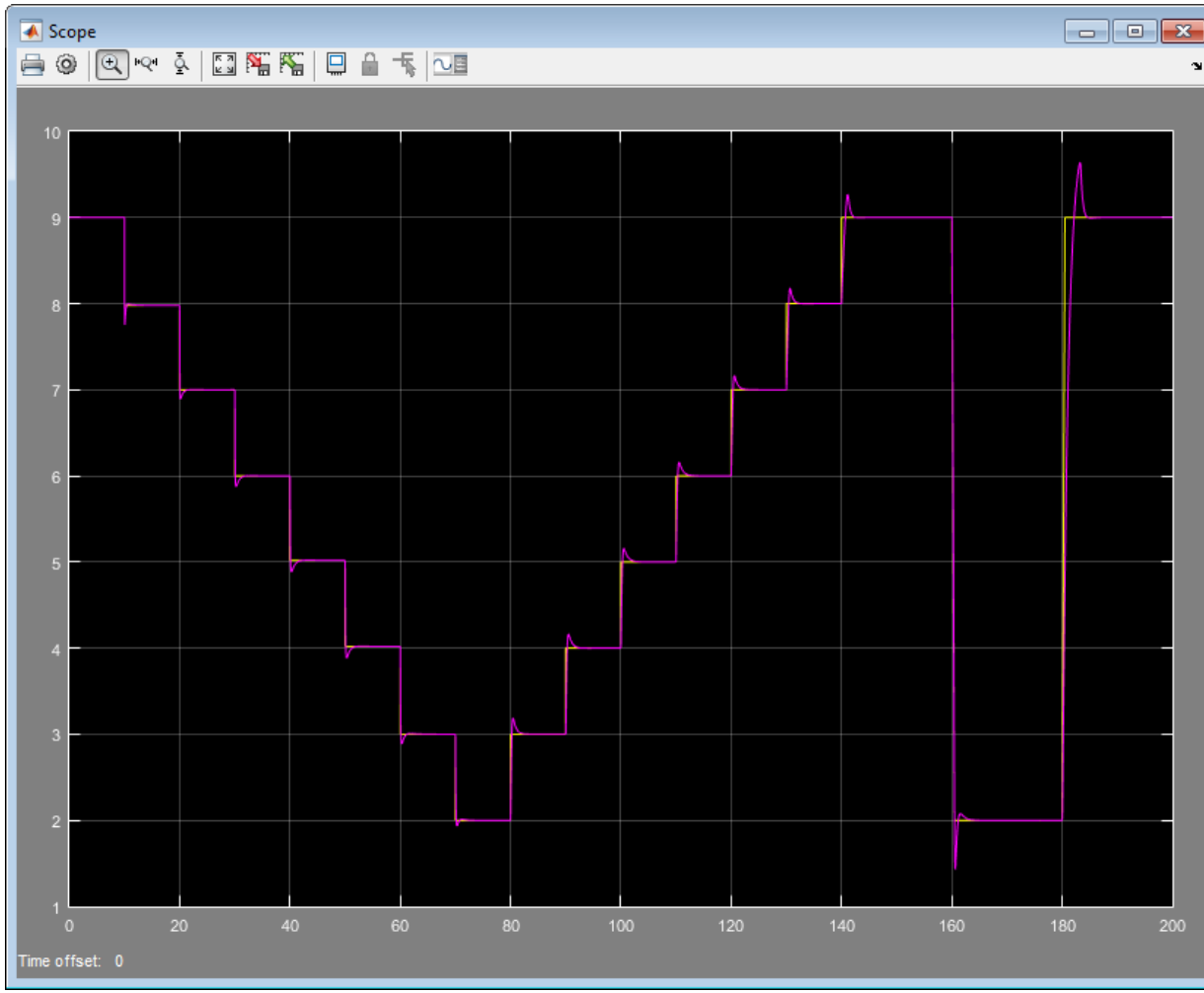
The **Table data** field contains the array of proportional coefficients for each controller, `Controllers.Kp`. (For more information about the properties of the `pid` models in the array `Controllers`, see the `pid` reference page.) Each entry in this array corresponds to an entry in the array `C` that is entered in the **Breakpoints 1** field. For concentration values that fall between entries in `C`, the `P Lookup Table` block performs linear interpolation to determine the value of the proportional coefficient. To set up lookup tables for the integral and derivative coefficients, configure the `I Lookup Table` and `D Lookup Table` blocks using `Controllers.Ki` and `Controllers.Kd`, respectively. For this example, this configuration is already done in the model.

The `pid` models in the `Controllers` array express the derivative filter coefficient as a time constant, `Controllers.Tf` (see the `pid` reference page for more information). However, the PID Controller block expresses the derivative filter coefficient as the inverse constant, `N`. Therefore, the `N Lookup Table` block must be configured to use the inverse of each value in `Controllers.Tf`. Double-click the `N Lookup Table` block to see the configuration.



Simulate the model. The `Concentration Setpoint` block is configured to step through a sequence of setpoints that spans the operating range between `C = 2` and `C = 9`

(shown in yellow on the scope). The simulation shows that the gain-scheduled configuration achieves good setpoint tracking across this range (pink on the scope).



As was shown in "Designing a Family of PID Controllers for Multiple Operating Points" on page 6-112, the CSTR plant is unstable in the operating range between `C = 4` and `C = 7`. The gain-scheduled PID controllers stabilize the plant and yield good setpoint tracking through the entire unstable region. To fully validate the control design against the nonlinear plant, apply a variety of setpoint test sequences that test the tracking

performance for steps of different size and direction across the operating range. You can also compare the performance against a design without gain scheduling, by setting all entries in the `Controllers` array equal.

See Also

PID Controller | n-D Lookup Table | `pid` | `pidtune`

More About

- “Designing a Family of PID Controllers for Multiple Operating Points” on page 6-112

Plant Cannot Be Linearized or Linearizes to Zero

When you open **PID Tuner**, it attempts to linearize the model at the operating point specified by the model initial conditions. Sometimes, **PID Tuner** cannot obtain a nonzero linear system for the plant as seen by the PID controller.

How to Fix It

If the plant model in the PID loop cannot be linearized or linearizes to zero, you have several options for obtaining a linear plant model for PID tuning. The following table summarizes some of the options and when they are useful.

Approach	Useful When	More Information
Linearize at a different operating point	There is a known operating point suitable for tuning, such as: • A simulation snapshot time at which the plant is in a linearizable steady state. • Known state values or a previously trimmed operating point at which the plant is linearizable.	"Tune at a Different Operating Point" on page 6-19
Import a linear model of the plant to **PID Tuner**	You have an LTI model of the plant at the desired operating condition for tuning in the MATLAB workspace.	In **PID Tuner**, in the **Plant** menu, select **Import**.
Tune the controller using simulated plant frequency-response data	The plant is not linearizable in any operating condition suitable for tuning.	"Design PID Controller from Plant Frequency-Response Data" on page 6-49

Approach	Useful When	More Information
Use system identification to estimate a linear plant model from measured or simulated response data	You have System Identification Toolbox software. An advantage of this approach is that it yields an analytic plant model that you can use for further analysis.	"Interactively Estimate Plant from Measured or Simulated Response Data" on page 6-73

See Also

More About

- "Cannot Find a Good Design in PID Tuner" on page 6-130
- "Introduction to Model-Based PID Tuning in Simulink" on page 6-3

Cannot Find a Good Design in PID Tuner

After adjusting the **PID Tuner** sliders, sometimes you cannot find a design that meets your design requirements when you analyze the **PID Tuner** response plots.

How to Fix It

Try a different PID controller type. It is possible that your controller type is not the best choice for your plant or your requirements.

For example, the closed-loop step response of a P- or PD-controlled system can settle on a value that is offset from the setpoint. If you require a zero steady-state offset, adding an integrator (using a PI or PID controller) can give better results.

As another example, in some cases a PI controller does not provide adequate phase margin. You can instead try a PID controller to give the tuning algorithm extra degrees of freedom to satisfy both speed and robustness requirements simultaneously.

To switch controller types, in the PID Controller block dialog box:

- Select a different controller type from the **Controller** drop-down menu.
- Click **Apply** to save the change.
- Click **Tune** to instruct **PID Tuner** to tune the parameters for the new controller type.

If you cannot find any satisfactory controller with **PID Tuner**, PID control possibly is not sufficient for your requirements. You can design more complex controllers using **Control System Designer**.

See Also

PID Controller

More About

- “Simulated Response Does Not Match the PID Tuner Response” on page 6-131
- “Control System Designer Tuning Methods” on page 8-6
- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3

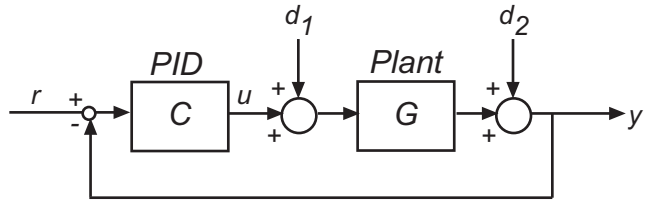
Simulated Response Does Not Match the PID Tuner Response

When you run your Simulink model using the PID gains computed by **PID Tuner**, the simulation output differs from the **PID Tuner** response plot.

There are several reasons why the simulated model can differ from the **PID Tuner** response plot. If the simulated result meets your design requirements (despite differing from the **PID Tuner** response), you do not need to refine the design further. If the simulated result does not meet your design requirements, see "Cannot Find an Acceptable PID Design in the Simulated Model" on page 6-133.

Some causes for a difference between the simulated and **PID Tuner** responses include:

- The reference signals or disturbance signals in your Simulink model differ from the step signals that **PID Tuner** uses. If you need step signals to evaluate the performance of the PID controller in your model, change the reference signals in your model to step signals.
- The structure of your model differs from the loop structure that **PID Tuner** designs for. **PID Tuner** assumes the loop configuration shown in the following figure.



  As the figure illustrates, **PID Tuner** designs for a PID controller in the feedforward path of a unity-gain feedback loop. If your Simulink model differs from this structure, or injects a disturbance signal in a different location, your simulated response differs from the **PID Tuner** response.
- You have enabled nonlinear features in the PID Controller block in your model, such as saturation limits or anti-windup circuitry. **PID Tuner** ignores nonlinear settings in the PID Controller block, which can cause **PID Tuner** to give a different response from the simulation.
- Your Simulink model has strong nonlinearities in the plant that make the linearization invalid over the full operating range of the simulation.
- You selected an operating point using **PID Tuner** that is different from the operating point saved in the model. In this case, **PID Tuner** has designed a controller for a

different operating point than the operating point that begins the simulation. Simulate your model using the **PID Tuner** operating point by initializing your Simulink model with this operating point. See “Simulate Simulink Model at Specific Operating Point” on page 1-83.

See Also

PID Controller

More About

- “Cannot Find an Acceptable PID Design in the Simulated Model” on page 6-133
- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3

Cannot Find an Acceptable PID Design in the Simulated Model

When you run your Simulink model using the PID gains computed by **PID Tuner**, the simulation output may not meet your design requirements.

How to Fix It

In some cases, PID control is not adequate to meet the control requirements for your plant. If you cannot find a design that meets your requirements when you simulate your model, consider designing a more complex controller using **Control System Designer**.

If you have enabled saturation limits in the PID Controller block without antiwindup circuitry, enable antiwindup circuitry. You can enable antiwindup circuitry in two ways:

- Activate the PID Controller block antiwindup circuitry on the **PID Advanced** tab of the block dialog box.
- Use the PID Controller block tracking mode to implement your own antiwindup circuitry external to the block. Activate the PID Controller block tracking mode on the **PID Advanced** tab of the block dialog box.

To learn more about both ways of implementing antiwindup circuitry, see “Anti-Windup Control Using a PID Controller” (Simulink).

After enabling antiwindup circuitry, run the simulation again to see whether controller performance is acceptable.

If the loop response is still unacceptable, try slowing the response of the PID controller. To do so, reduce the response time or the bandwidth in **PID Tuner**. See “Refine the Design” on page 6-16.

You can also try implementing gain-scheduled PID control to help account for nonlinearities in your system. See “Designing a Family of PID Controllers for Multiple Operating Points” on page 6-112 and “Implement Gain-Scheduled PID Controllers” on page 6-121.

If you still cannot get acceptable performance with PID control, consider using a more complex controller. See **Control System Designer**.

See Also

PID Controller

More About

- “Simulated Response Does Not Match the PID Tuner Response” on page 6-131
- “Controller Performance Deteriorates When Switching Time Domains” on page 6-135
- “Control System Designer Tuning Methods” on page 8-6
- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3

Controller Performance Deteriorates When Switching Time Domains

After you obtain a well-tuned, continuous-time controller using **PID Tuner**, you can discretize the controller using the **Time Domain** selector button in the PID Controller block dialog box. Sometimes, the resulting discrete-time controller performs poorly or even becomes unstable.

How To Fix It

In some cases, you can improve performance by adjusting the sample time by trial and error. However, this procedure can yield a poorly tuned controller, especially where your application imposes a limit on the sample time. Instead, if you change time domains and the response deteriorates, click **Tune** in the PID Controller block dialog to design a new controller.

Note If the plant and controller time domains differ, **PID Tuner** discretizes the plant (or converts the plant to continuous time) to match the controller time domain. If the plant and controller both use discrete time, but have different sample times, **PID Tuner** resamples the plant to match the controller. All conversions use the `tustin` method (see "Continuous-Discrete Conversion Methods" (Control System Toolbox).

See Also

PID Controller

More About

- "When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain" on page 6-136
- "Introduction to Model-Based PID Tuning in Simulink" on page 6-3

When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain

When you design a controller using **PID Tuner**, the resulting derivative gain, *D*, can have a different sign from the integral gain *I*. **PID Tuner** always returns a stable controller, even if one or more gains are negative.

For example, the following expression gives the PID controller transfer function in `Ideal` form:

$$c = P\left(1 + \frac{I}{s} + \frac{Ds}{\frac{s}{N} + 1}\right) = P\frac{(1 + DN)s^2 + (I + N)s + IN}{s(s + N)}$$

For a stable controller, all three numerator coefficients require positive values. Because N is positive, $IN > 0$ requires that I is also positive. However, the only restriction on D is $(1 + DN) > 0$. Therefore, as long as $DN > -1$, a negative D still yields a stable PID controller.

Similar reasoning applies for any controller type and for the `Parallel` controller form. For more information about controller transfer functions, see the PID Controller block reference page.

See Also

PID Controller

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 6-3

PID Autotuning

PID Autotuning Basics

The real-time PID autotuning tools in Simulink Control Design let you deploy an automatic tuning algorithm to tune a controller against a physical plant. Real-time PID autotuning lets you tune a PID controller to achieve a specified bandwidth and phase margin without a parametric plant model or an initial controller design.

To achieve model-free tuning, use the Online PID Tuner block. This block operates as follows:

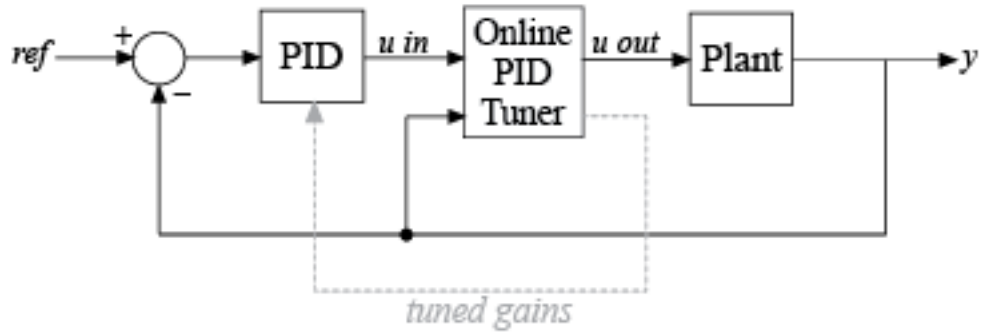
1. Injects a test signal into the plant at the nominal operating point to collect plant input-output data and estimate frequency response in real time. The test signal is a combination of sine and step perturbation signals added on top of the nominal plant input measured when the experiment starts. If the plant is part of a feedback loop, the block opens the loop during the experiment.
2. At the end of the experiment, tunes PID controller parameters based on estimated plant frequency responses near the open-loop bandwidth.
3. Updates a PID Controller block or a custom PID controller with the tuned parameters, allowing you to validate closed-loop performance in real time.

You can generate code that implements the tuning algorithm on hardware, letting you tune with or without Simulink in the loop. (Doing so requires a code-generation product such as Simulink Coder™.)

To access the block, in the Simulink Library Browser, select **Simulink Control Design**.

How PID Autotuning Works

The following schematic diagram illustrates how you incorporate PID autotuning into your system.



You insert the PID autotuning module (Online PID Tuner) into your system such that it accepts a control signal and feeds into the plant. You can initiate the PID autotuning process at any time. Until the the autotuning process begins, Online PID Tuner relays the control signal directly from `u in` to `u out`. In that state, the module has no effect on the performance of your system.

When the autotuning process begins, Online PID Tuner first performs a frequency-response estimation experiment. Online PID Tuner breaks the feedback loop between `u in` to `u out`. It then injects into `u out` a superposition of sinusoidal test signals at four frequencies, [1/3, 1, 3, 10]ω_c, where ω_c is the target bandwidth for tuning that you specify. Online PID Tuner uses the response to these test signals at `y` to estimate the plant frequency response at the nominal operating point. If the plant is asymptotically stable, the experiment can also include an estimation of the plant DC gain obtained by injecting a step test signal.

When the experiment ends, the PID autotuning algorithm uses the estimated frequency response to compute PID gains. The algorithm aims to balance performance and robustness while achieving the control bandwidth and minimum phase margin that you specify in the block. You can examine the tuned gains and transfer them to the PID controller.

To use the algorithm, you do not need an initial PID controller design. However, you must have some way to get the plant to a nominal operating point for the frequency-response estimation experiment.

When to Use PID Autotuning

Embedded PID autotuning is a useful option when you have a PID-controlled system and a test bed or control environment to operate in. In this case, you can deploy the Online PID Tuner block to add a module that can automatically tune the gains of the PID controller in your system.

In practice, you can implement the illustrated system in several ways, including:

- Deploy the autotuning algorithm as a standalone embedded module on your hardware, removing Simulink from the loop. For more information, see "PID Autotuning in Real Time" on page 7-6.
- Initiate, monitor, and analyze the autotuning process via Simulink, and then apply the tuned parameters and deploy the PID controller. To control the autotuning process in Simulink, you run a Simulink model that contains the Online PID Tuner block in external simulation mode. For more information, see "PID Autotuning in External Mode" on page 7-14.

PID autotuning works with any asymptotically stable SISO plant, whether low-order or high-order, with or without time delay, and with or without direct feedthrough. It can tune any type of PID controller. You trigger the tuning process via an input to the Online PID Tuner block, so you can tune and retune your controller at any time.

Caution Because the Online PID Tuner block performs an open-loop estimation experiment, do not use the block with an unstable plant or a plant with multiple integrators.

If you do have a plant model in Simulink, you can use PID autotuning to obtain an initial PID design. Doing so lets you preview plant response and adjust the settings for PID autotuning before tuning the controller in real time.

See Also

Online PID Tuner

More About

- "PID Autotuning in Real Time" on page 7-6

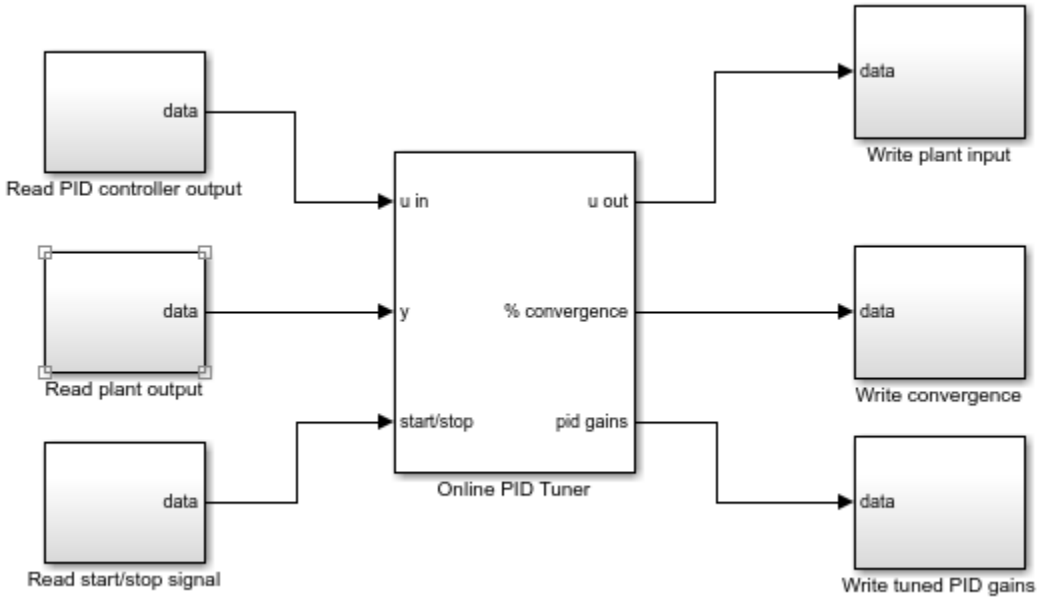
- “PID Autotuning in External Mode” on page 7-14

PID Autotuning in Real Time

To use the PID autotuning algorithm for real-time tuning in a standalone application, you must deploy the Online PID Tuner block into your own system. In your system, it injects signals into your plant and receives the plant response. To do so, you create a Simulink model for deployment. You can configure this model with the experiment and tuning parameters. Or, you can configure it to supply such parameters externally from elsewhere in your system. (Deploying the PID autotuning algorithm requires a code-generation product such as Simulink Coder.)

Simulink Models for Deploying Autotuning Algorithm

In the most basic form, a model for deploying real-time PID autotuning resembles the following illustration.

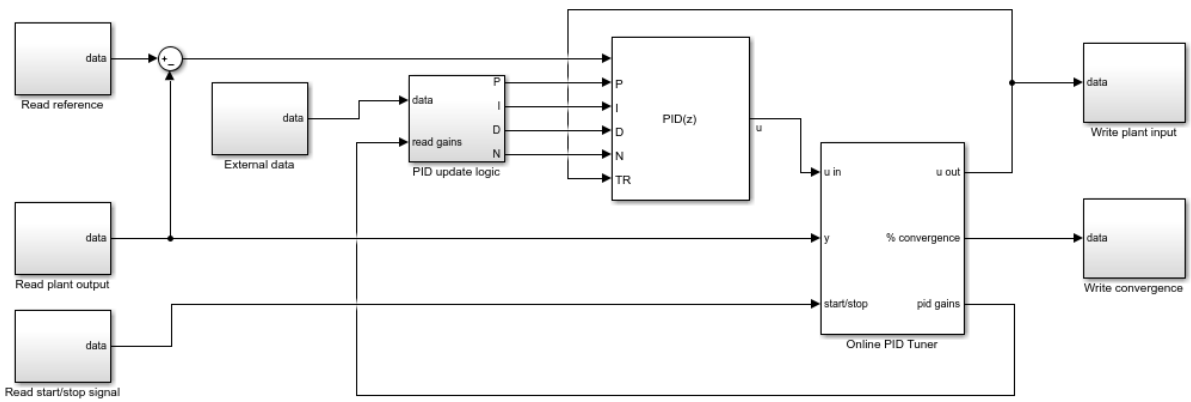


Here, the blocks connected to the inputs and outputs of the Online PID Tuner block represent hardware interfaces that read or write data for your system. The default ports of the block are:

- `u in` — Receives the control signal.
- `y` — Receives the plant output.
- `start/stop` — Receives the signal that begins and ends the tuning process.
- `u out` — Outputs the signal to feed to the plant input. When the experiment is not running, `u out` outputs the control signal as input at `u in`. When the experiment is running, the block breaks the loop between `u in` and `u out`, and injects the test signals at `u out`.
- `% convergence` — Outputs a numeric indicator of the progress of the frequency-response estimation experiment.
- `pid gains` — Outputs the tuned PID gains when the tuning process stops.

In this configuration, the PID controller itself exists in another module of your system. When tuning is complete, you write the tuned PID gains from the `pid gains` port to the PID controller.

Alternatively, you can deploy a module that includes both the PID controller and the PID autotuning algorithm, such as shown in the following illustration.



In this illustration, the PID controller is implemented as a Simulink PID Controller block. Because the PID gains of that block are tunable, you can configure your system to write the tuned gains to the deployed controller. Alternatively, you can also use your own custom PID controller subsystem in the model that you deploy. You can implement any logic appropriate to your application to determine whether and how to update the PID controller with the tuned gains. In the illustrated system, the `PID update logic`

subsystem represents such a module. The `External data` block represents whatever other information your logic requires to determine whether to update the controller.

When you transfer the PID gains to your own controller, be aware of the meaning of these gains in the Online PID Tuner block. In discrete time, the PID controller transfer function assumed by the block is

$$C = P + F_i(z)I + \frac{D}{N + F_d(z)},$$

in parallel form, or

$$C = P\left(1 + \frac{F_i(z)}{I} + \frac{D}{D/N + F_d(z)}\right).$$

in ideal form. $F_i(z)$ and $F_d(z)$ depend on the values you specify for the **Integrator method**and **Filter method** formulas, respectively. For more details, see the Online PID Tuner block reference page.

Caution When your PID controller includes integrator action, consider implementing signal tracking to avoid integrator windup during the tuning experiment. If your PID controller is a Simulink PID Controller block, use the **Enable tracking mode** parameter of the controller block to do so. If you are using a PID controller without signal tracking, be aware of the potential for integrator wind-up or for a bump when control is transferred.

Access Autotuning Parameters After Deployment

To use the PID autotuning algorithm in your system, you must configure certain parameters of the Online PID Tuner block that govern the estimation experiment and tuning goals. Some of these parameters are tunable, such that you can access them in the generated code. For the parameters that are not tunable, you must configure them in the Online PID Tuner before deployment.

Tunable Parameters

The following parameters of the Online PID Tuner block are tunable after deployment. For more information about all these parameters, see the Online PID Tuner block reference page.

Parameter	Description
Target bandwidth (rad/sec)	Target crossover frequency of open-loop response
Target phase margin (degrees)	Target minimum phase margin of open-loop response
Sine Amplitudes	Amplitude of sinusoidal perturbations
Estimate DC gain with step signal	Inject step signal into plant
Step Amplitude	Amplitude of step perturbation
Type	PID controller type (such as PI, PD, or PID)
Form	PID controller form
Integrator method	Discrete integration formula for integrator term
Filter method	Discrete integration formula for derivative filter term

Non-Tunable Parameters

The following parameters of the Online PID Tuner block are not tunable after deployment. You specify them in the Online PID Tuner before code generation, and their values remain fixed in your application. For more information about all these parameters, see the Online PID Tuner block reference page.

Parameter	Description
Time Domain	PID controller time domain
Controller sample time (sec)	Sample time of PID controller (see “Modify Sample Times After Deployment” on page 7-9)
Experiment sample time (sec)	Sample time for experiment
Decrease memory footprint (external mode only)	Deploy tuning algorithm only
Data Type	Floating point precision

Modify Sample Times After Deployment

The **Controller sample time (sec)** parameter is not tunable. As a consequence, you cannot access them directly in generated code when you deploy the Online PID Tuner

block. However, you can run the deployed block with different sample times in your application. To do so:

1. Set **Controller sample time (sec)** to –1.
2. Put the Online PID Tuner block in a Triggered Subsystem.
3. Trigger the subsystem at the desired sample time.

Configure Start/Stop Signal

To start and stop the online tuning process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts. When the experiment is running, the block opens the loop between `u in` and `u out` and injects test signals at `u out`.
- Positive to negative or zero, the experiment stops. When the experiment is not running, the block passes signals unchanged from `u in` to `u out`. In this state, the block has no impact on plant or controller behavior.

Typically, you can control the experiment with a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. Some points to consider when configuring the **start/stop** signal include:

- Start the experiment when the plant is at the desired equilibrium operating point. If necessary, you can use the source block connected to **u in** to drive the plant to the operating point.
- Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.
- Avoid stopping the experiment prematurely. You must let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. There are two ways to determine when stop the experiment:
  - Determine the experiment duration in advance. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning.
  - Observe the signal at the `convergence` output, and stop the experiment when the signal stabilizes near 100%.
- When the experiment stops, the Online PID Tuner block computes tuned PID gains and updates the signal at the `pid gains` port.

Set PID Tuning Parameters

In the Online PID Tuner block or in your deployed application, specify the configuration of the PID controller in your system. To do so, use the following block parameters:

- **Type**
- **Form**
- **Time Domain**
- **Controller sample time (sec)**
- **Integrator method**
- **Filter method**

For more information about setting these parameters, see the Online PID Tuner block reference page.

You also specify your tuning goals in the block or in your application. Specify the target bandwidth and phase margin for tuning with the **Target bandwidth (rad/sec)** and **Target phase margin (degrees)** parameters, respectively.

The target bandwidth is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ, a good guess for the target bandwidth is $2/\tau$.

To perform PID tuning, the Online PID Tuner block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s is the controller sample time that you specify with the **Controller sample time (sec)** parameter. Because of this condition, the fastest rise time you can enforce for tuning is about $1.67T_s$. If this rise time does not meet your design goals, consider reducing T_s.

The target phase margin reflects your desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

Set Experiment Parameters

In the block or in your application, you also specify the parameters of the frequency-response estimation experiment.

The experiment injects sinusoidal signals at four frequencies near the target bandwidth, $[1/3, 1, 3, 10]\omega_c$. Use the **Sine Amplitudes** parameter to specify the amplitudes of these signals. You can provide a scalar value to inject the same amplitude at each frequency, or a vector of length 4 to specify different amplitudes for each.

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the four frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that there is a sharp decay in response over the range of frequencies, consider decreasing the amplitude of $(1/3)\omega_c$ and increasing the amplitude of $10\omega_c$. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level.
- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output.

In the experiment, the four sinusoidal signals are superimposed (with the step perturbation, if any). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

- Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
- How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID controller used in engine-speed control. You have determined that a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

If your plant is asymptotically stable, the experiment can also use a step signal perturbation to estimate the plant DC gain. Specify the amplitude of this perturbation

with the **Step Amplitude** parameter. The considerations for choosing a step amplitude are the same as the considerations for specifying the amplitudes of the sinusoidal perturbations. If your plant has a single integrator, clear the **Estimate DC gain with step signal** parameter.

Caution Because of the open-loop nature of the estimation experiment, do not use PID autotuning with an unstable plant or a plant with multiple integrators.

See Also

Online PID Tuner

More About

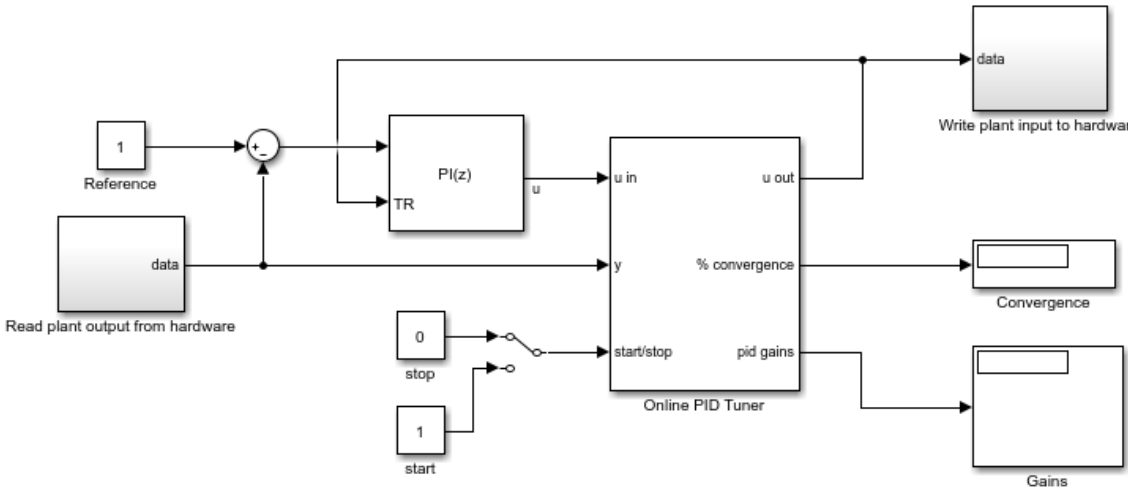
- "PID Autotuning Basics" on page 7-2
- "PID Autotuning in External Mode" on page 7-14

PID Autotuning in External Mode

Before or instead of deploying the PID autotuning algorithm as described in "PID Autotuning in Real Time" on page 7-6, you can run the algorithm on hardware while controlling the experiment from Simulink. To do so, you use a model that contains a PID controller and the Online PID Tuner block, and run this model in external mode. External mode allows communication between the Simulink block diagram and the standalone program that is built from the generated code. In this mode, Simulink serves as a real-time monitoring interface in which you can interact with the tuning algorithm running on hardware. For instance, you can start and stop the experiment or change tuning goals from the Simulink interface while the model is running. (Running the PID autotuning algorithm in external mode requires a code-generation product such as Simulink Coder.)

Simulink Models for PID Autotuning in External Mode

A model for tuning PID autotuning in external mode resembles the following illustration.



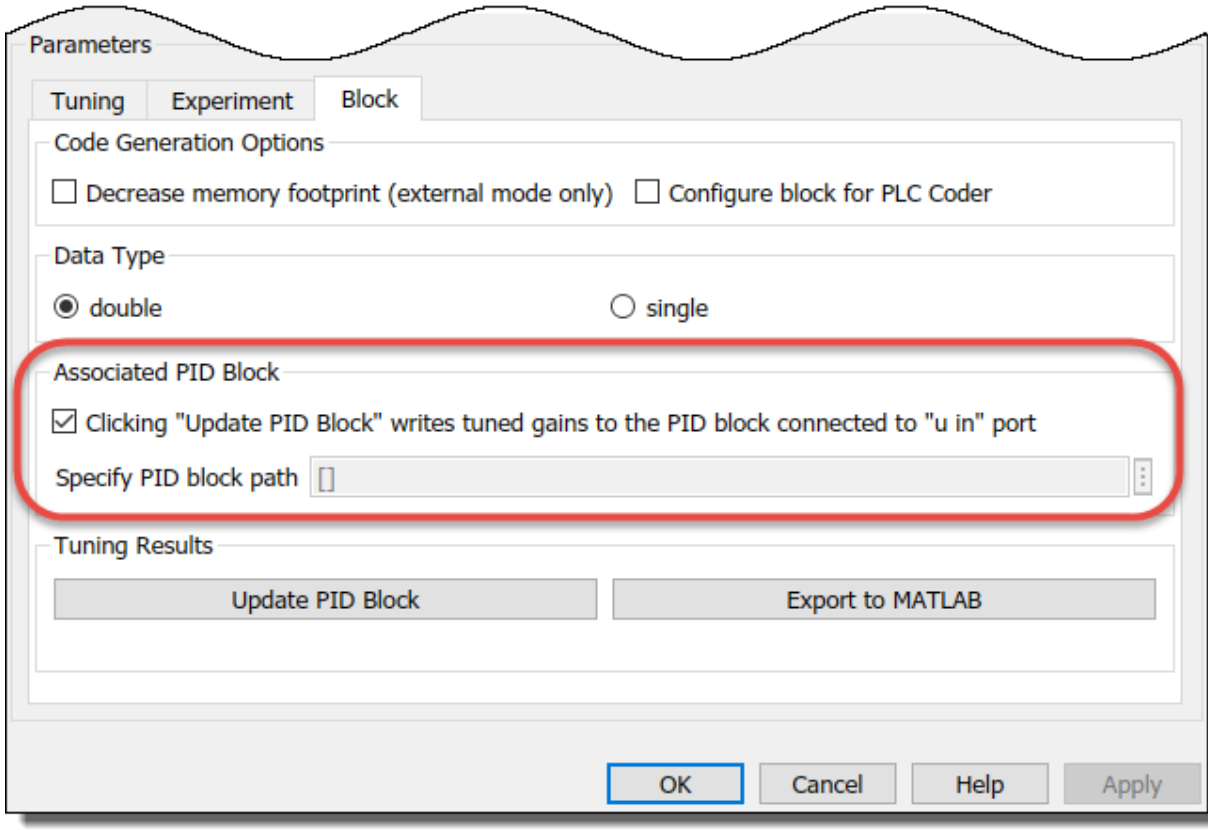
Here, the blocks marked `Read plant output` and `Write control signal` represent hardware interfaces that read or write data for your system. The PID controller block that you use can be either of the following:

- A Simulink PID Controller block. In this case, you can write tuned controller parameters directly from the Online PID Tuner block to the PID controller on hardware in real time. To do so, when the tuning process ends, in the Online PID Tuner block, click **Update PID Tuner**.
- A custom PID controller. In this case, you can write tuned controller parameters directly from the Online PID Tuner block if the following conditions are both true:

  - The custom controller is a masked subsystem.
  - The PID gains are mask parameters named `P`, `I`, `D`, and `N`. (You do not need to use all four parameters. For example, if you use a custom PI controller, then you only need mask parameters `P` and `I`.)

  If your custom PID controller does not meet these conditions, you must transfer the tuned gains to your controller some other way, such as manually or with your own logic.

For the Online PID Tuner to write the tuned PID gains to your PID controller, you must designate your controller as the associated PID block for the Online PID Tuner block. You do so with the block parameters in the **Associated PID Block**. For the configuration in the illustration, selecting **Clicking "Update PID Block" writes tuned gains to the PID block connected to "u in" port** is sufficient. If your PID controller does not feed directly into the Online PID Tuner block, clear that option, and specify the block path to your PID controller block.



Caution When your PID controller includes integrator action, consider implementing signal tracking to avoid integrator windup during the tuning experiment. If your PID controller is a Simulink PID Controller block, use the **Enable tracking mode** parameter of the controller block to do so. If you are using a PID controller without signal tracking, be aware of the potential for integrator wind-up or for a bump when control is transferred.

Configure Start/Stop Signal

To start and stop the online tuning process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts. When the experiment is running, the block opens the loop between `u in` and `u out` and injects test signals at `u out`.
- Positive to negative or zero, the experiment stops. When the experiment is not running, the block passes signals unchanged from `u in` to `u out`. In this state, the block has no impact on plant or controller behavior.

Typically, you can control the experiment with a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. Some points to consider when configuring the **start/stop** signal include:

- Start the experiment when the plant is at the desired equilibrium operating point. If necessary, you can use the source block connected to **u in** to drive the plant to the operating point.
- Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.
- Avoid stopping the experiment prematurely. You must let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. There are two ways to determine when stop the experiment:
  - Determine the experiment duration in advance. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning.
  - Observe the signal at the `convergence` output, and stop the experiment when the signal stabilizes near 100%.
- When the experiment stops, the Online PID Tuner block computes tuned PID gains and updates the signal at the `pid gains` port.

When you control the tuning in Simulink, you can control the start and end of the experiment manually. For instance, in the illustrated system, the start/stop signal is a simple switch. When the model is running in external mode, you toggle the switch to 1 to begin the experiment. After an appropriate time, or after the `% convergence` signal settles near 100, toggle the switch again to end the experiment. When the experiment ends, the algorithm generates the tuned PID gains.

Alternatively, you can configure the `start/stop` signal to begin and end the experiment automatically at particular simulation times. For example, you can use:

- A Pulse Generator block — Configure the pulse generator to start the pulse after some delay which is long enough for the plant to reach a steady-state operating point.

Configure it to stop the pulse after about $100/\omega_c$. An advantage of this approach is that you can control the experiment start and duration in a single block. However, it a good practice to set the simulation time of your model to be only a little larger than the duration of the pulse plus the delay. Doing so avoids restarting the experiment while the simulation is running.

- The sum of two Step blocks — Configure one Step block to step from 0 to 1 at the desired experiment start time. Configure a second Step block to step from 1 to 0 at the desired end time. Feed the sum of the two signals into the `start/stop` port of the Online PID Tuner block. An advantage of this approach is that it is easy to ensure that the experiment only runs once during simulation.

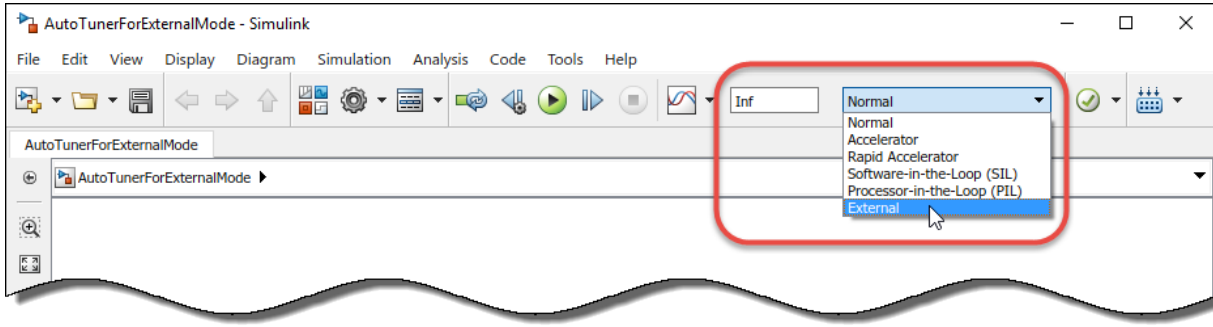
You can configure any other logic appropriate for your application to control the start and stop times of the experiment.

Configure Tuning Parameters and Experiment Parameters

In the Online PID Tuner block, you specify your tuning goals such as target bandwidth and phase margin. You also configure parameters of the experiment, such as the amplitudes of the sinusoidal perturbation. You configure these parameters as described in “PID Autotuning in Real Time” on page 7-6.

Run the Model and Tune the Controller

After configuring the parameters of the Online PID Tuner block, select `external mode`, set the simulation time to infinite, and run the model.



Simulink compiles the model and deploys it to your connected hardware.

- If you have configured the `start/stop` signal to begin and end the tuning process at specific times, allow the simulation to run through the end of the experiment.
- If you have configured a manual `start/stop` signal, begin the experiment when your plant has reached steady-state. Observe the signal at the `convergence` output, and stop the experiment when the signal stabilizes near 100%.

Validate Tuning Results

When the experiment ends, the Online PID Tuner block computes new PID gains based on the estimated frequency response at the system and your specified tuning goals. Examine them for reasonableness. For instance, if you have an initial PID controller, you might expect the tuned gains to be roughly the same magnitude as the gains of the initial design. There are several ways to see the tuned gains:

- View the output of the `pid gains` port of the block. One way to view this output is to connect the output to a Simulink Display block.
- In the Online PID Tuner block, in the **Block** tab, click **Export to MATLAB**. The block creates a structure in the MATLAB workspace, `OnlinePIDTuningResult`. This structure contains fields `P`, `I`, `D`, and `N` (whichever of these gains are needed for the PID type you are tuning) that contain the tuned gain values. (For more information about the contents of this structure, see the Online PID Tuner block reference page.)

The Online PID Tuner block does not automatically push the tuned gains to the associated PID block. To do so, in the Online PID Tuner block, in the **Block** tab, click **Update PID Block**. You can update the PID block while the simulation is running, including when running in external mode. Doing so is useful for immediately validating tuned PID gains.

Note While running In external mode, you can change parameters at any time during simulation, start the experiment again, and push the new tuned gains to the PID block. You can then continue to run the model and observe the behavior of your plant.

For an example illustrating PID autotuning in external mode, see “Tune PID Controller in Real Time Using Online PID Tuner Block”.

See Also

Online PID Tuner

More About

- “PID Autotuning in Real Time” on page 7-6
- “PID Autotuning Basics” on page 7-2

Classical Control Design

Choose a Control Design Approach

Simulink Control Designlets you design and tune many types of control systems in Simulink. There are are also deployable PID autotuning tools that let you tune your controller in real time against a physical plant.

Design in Simulink

Simulink Control Design provides several approaches to tuning Simulink blocks, such as Transfer Fcn and PID Controller blocks. Use the following table to determine which approach best supports what you want to do.

	Model-Based PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Supported Blocks	PID Controller PID Controller 2DOF	Linear Blocks (see "What Blocks Are Tunable?" on page 8-12)	Any blocks; only some blocks are automatically parameterized (See "How Tuned Simulink Blocks Are Parameterized" (Control System Toolbox))
Architecture	1-DOF and 2-DOF PID loops	Control systems that contain one or more SISO compensators	Any structure, including any number of SISO or MIMO feedback loops

	Model-Based PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Control Design Approach	Automatically tune PID gains to balance performance and robustness	• Graphically tune poles and zeros on design plots, such as Bode, root locus, and Nichols • Automatically tune compensators using response optimization (Simulink Design Optimization), LQG synthesis, or IMC tuning	Automatically tune controller parameters to meet design requirements you specify, such as setpoint tracking, stability margins, disturbance rejection, and loop shaping (see "Tuning Goals" (Control System Toolbox))
Analysis of Control System Performance	Time and frequency responses for reference tracking and disturbance rejection	Any combination of system responses	Any combination of system responses

	Model-Based PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Interface	• Graphical tuning using **PID Tuner** (see "Introduction to Model-Based PID Tuning in Simulink" on page 6-3) • Tuning of plants that do not linearize (see "Frequency Response Based Tuning Basics" on page 6-51) • Programmatic tuning using `pidtune` (see "PID Controller Design at the Command Line" (Control System Toolbox))	Graphical tuning using **Control System Designer**	• Graphical tuning using **Control System Tuner** • Programmatic tuning using `slTuner` (see "Programmatic Tuning")

Real-Time PID Autotuning

The real-time PID autotuning tools in Simulink Control Design let you deploy an automatic tuning algorithm as a stand-alone application for PID tuning against a physical plant. Real-time PID autotuning lets you tune a PID controller to achieve a specified bandwidth and phase margin without a parametric plant model or an initial controller design.

The real-time PID autotuning algorithm can tune PID gains in Simulink PID Controller blocks or in your own custom PID blocks. You can tune against your physical plant with or without Simulink in the loop. Deploying the real-time PID autotuning algorithm requires a code-generation product such as Simulink Coder.

For more information, see "PID Autotuning Basics" on page 7-2.

See Also

More About

- “PID Controller Tuning”
- “Classical Control Design”
- “Tuning with Control System Tuner”
- “Programmatic Tuning”

Control System Designer Tuning Methods

Using **Control System Designer**, you can tune compensators using various graphical and automated tuning methods.

In this section...

Graphical Tuning Methods

Use graphical tuning methods to interactively add, modify, and remove controller poles, zeros, and gains.

Tuning Method	Description	Useful For
Bode Editor	Tune your compensator to achieve a specific open-loop frequency response (loop shaping).	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.
Closed-Loop Bode Editor	Tune your prefilter to improve closed-loop system response.	Improving reference tracking, input disturbance rejection, and noise rejection.
Root Locus Editor	Tune your compensator to produce closed-loop pole locations that satisfy your design specifications.	Designing to time-domain design specifications, such as maximum overshoot and settling time.
Nichols Editor	Tune your compensator to achieve a specific open-loop response (loop shaping), combining gain and phase information on a Nichols plot.	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.

When using graphical tuning, you can modify the compensator either directly from the editor plots or using the compensator editor. A common design approach is to roughly tune your compensator using the editor plots, and then use the compensator editor to

fine-tune the compensator parameters. For more information, see “Edit Compensator Dynamics” (Control System Toolbox)

The graphical tuning methods are not mutually exclusive. For example, you can tune your compensator using both the Bode editor and root locus editor simultaneously. This option is useful when designing to both time-domain and frequency-domain specifications.

For examples of graphical tuning, see the following:

- “Bode Diagram Design” (Control System Toolbox)
- “Root Locus Design” (Control System Toolbox)
- “Nichols Plot Design” (Control System Toolbox)

Automated Tuning Methods

Use automated tuning methods to automatically tune compensators based on your design specifications.

Tuning Method	Description	Requirements and Limitations
PID Tuning	Automatically tune PID gains to balance performance and robustness or tune controllers using classical PID tuning formulas.	Classical PID tuning formulas require a stable or integrating effective plant.
Optimization Based Tuning	Optimize compensator parameters using design requirements specified in graphical tuning and analysis plots.	Requires Simulink Design Optimization software. Tunes the parameters of a previously defined controller structure.
LQG Synthesis	Design a full-order stabilizing feedback controller as a linear-quadratic-Gaussian (LQG) tracker.	Maximum controller order depends on the effective plant dynamics.

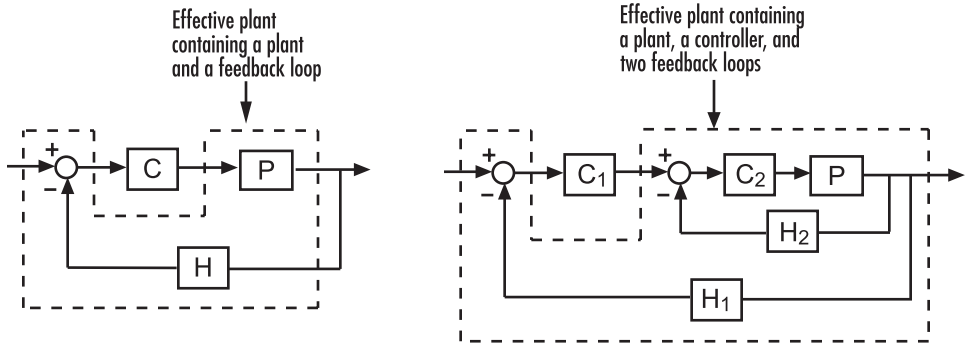
Tuning Method	Description	Requirements and Limitations
Loop Shaping	Find a full-order stabilizing feedback controller with a specified open-loop bandwidth or shape.	Requires Robust Control Toolbox software. Maximum controller order depends on the effective plant dynamics.
Internal Model Control (IMC) Tuning	Obtain a full-order stabilizing feedback controller using the IMC design method.	Assumes that your control system uses an IMC architecture that contains a predictive model of your plant dynamics. Maximum controller order depends on the effective plant dynamics.

A common design approach is to generate an initial compensator using PID tuning, LQG synthesis, loop shaping, or IMC tuning. You can then improve the compensator performance using either optimization-based tuning or graphical tuning.

For more information on automated tuning methods, see “Design Compensator Using Automated Tuning Methods” (Control System Toolbox).

Effective Plant for Tuning

An *effective plant* is the system controlled by a compensator that contains all elements of the open loop in your model other than the compensator you are tuning. The following diagrams show examples of effective plants:



Knowing the properties of the effective plant seen by your compensator can help you understand which tuning methods work for your system. For example, some automated tuning methods apply only to compensators whose open loops ($L = C\hat{P}$) have stable effective plants ($\hat{P}$). Also, for tuning methods such as IMC and loop shaping, the maximum controller order depends on the dynamics of the effective plant.

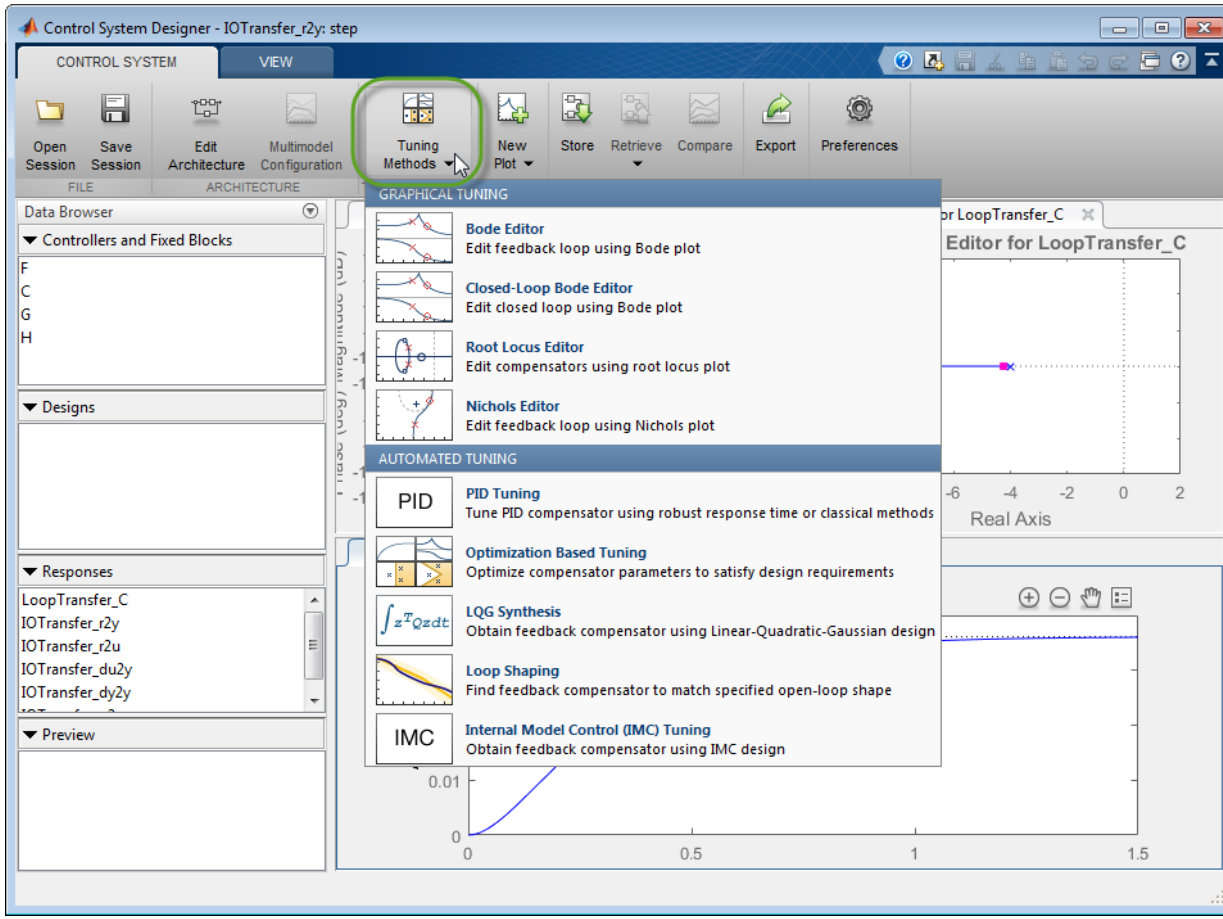
Tuning Compensators In Simulink

If the compensator in your Simulink model has constraints on its poles, zeros, or gain, you cannot use LQG synthesis, loop shaping, or IMC tuning. For example, you cannot tune the parameters of a PID Controller block using these methods. If your application requires controller constraints, use an alternative automated or graphical tuning method.

Also, any compensator constraints in your Simulink model limit the structure of your tuned compensator. For example, if you are using PID tuning and you configure your PID Controller block as a PI controller, your tuned compensator must have a zero derivative parameter.

Select a Tuning Method

To select a tuning method, in **Control System Designer**, click **Tuning Methods**.



See Also

Control System Designer

Related Examples

- “Bode Diagram Design” (Control System Toolbox)
- “Root Locus Design” (Control System Toolbox)
- “Nichols Plot Design” (Control System Toolbox)

- “Design Compensator Using Automated Tuning Methods” (Control System Toolbox)

What Blocks Are Tunable?

You can tune parameters in the following Simulink blocks using Simulink Control Design software. The block input and output signals for tunable blocks must have scalar, double-precision values.

Tunable Block	Description
Gain	Constant gain
LTI System	Linear time-invariant system
Discrete Filter	Discrete-time infinite impulse response filter
PID Controller	One degree-of-freedom PID controller
State-Space	Continuous-time state-space model
Discrete State-Space	Discrete-time state-space model
Zero-Pole	Continuous-time zero-pole-gain transfer function
Discrete Zero-Pole	Discrete-time zero-pole-gain transfer function
Transfer Fcn	Continuous-time transfer function model
Discrete Transfer Fcn	Discrete-time transfer function model

Additionally, you can tune the linear State-Space, Zero-Pole, and Transfer Fcn blocks in the Simulink Extras Additional Linear library.

You can tune the following versions of the listed tunable blocks:

- Blocks with custom configuration functions associated with a masked subsystem
- Blocks discretized using the Simulink Model Discretizer

Note If your model contains Model blocks with normal-mode model references to other models, you can select tunable blocks in the referenced models for compensator design.

See Also

Control System Designer

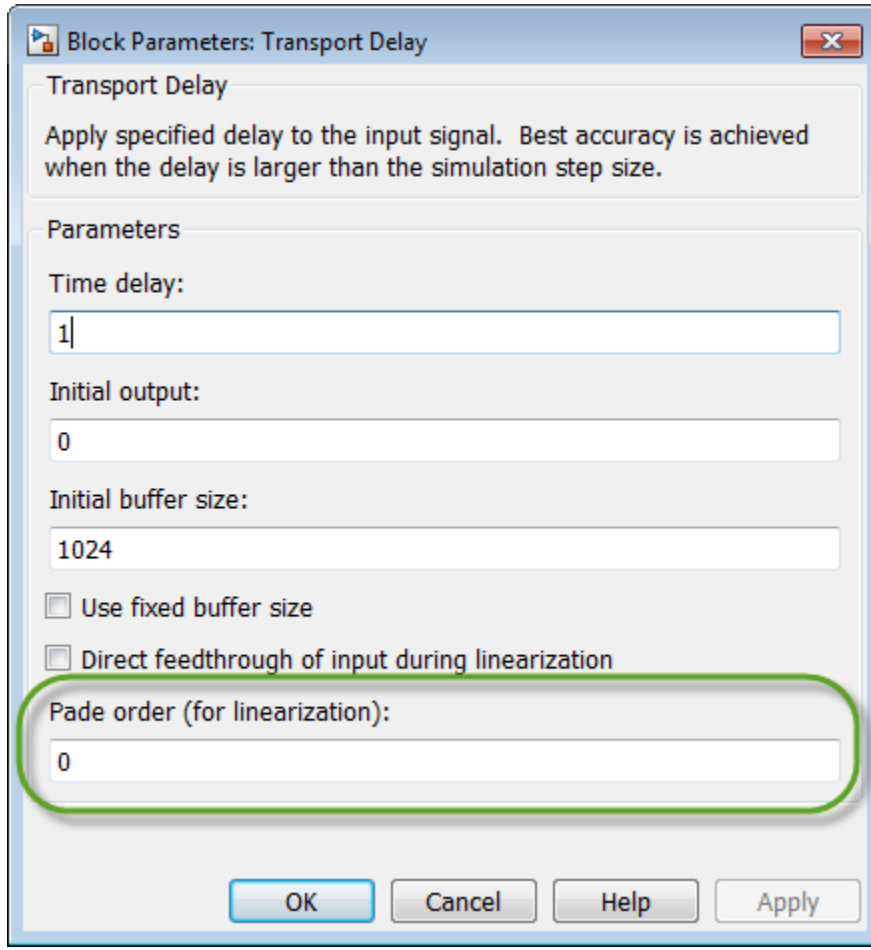
Related Examples

- “Control System Designer Tuning Methods” on page 8-6

Designing Compensators for Plants with Time Delays

You can design compensators for plants with time delays using Simulink Control Design software. When opened from Simulink, **Control System Designer** creates a linear model of your plant. Within this model, you can represent time delays using either Padé approximations or exact delays.

To represent time delays using Padé approximations, specify the Padé order in the Block Parameters dialog box for each Simulink block with delays.



If you do not specify a Padé order for a block, **Control System Designer** uses exact delays by default when possible.

However, the following design methods and plots do not support systems with exact time delays:

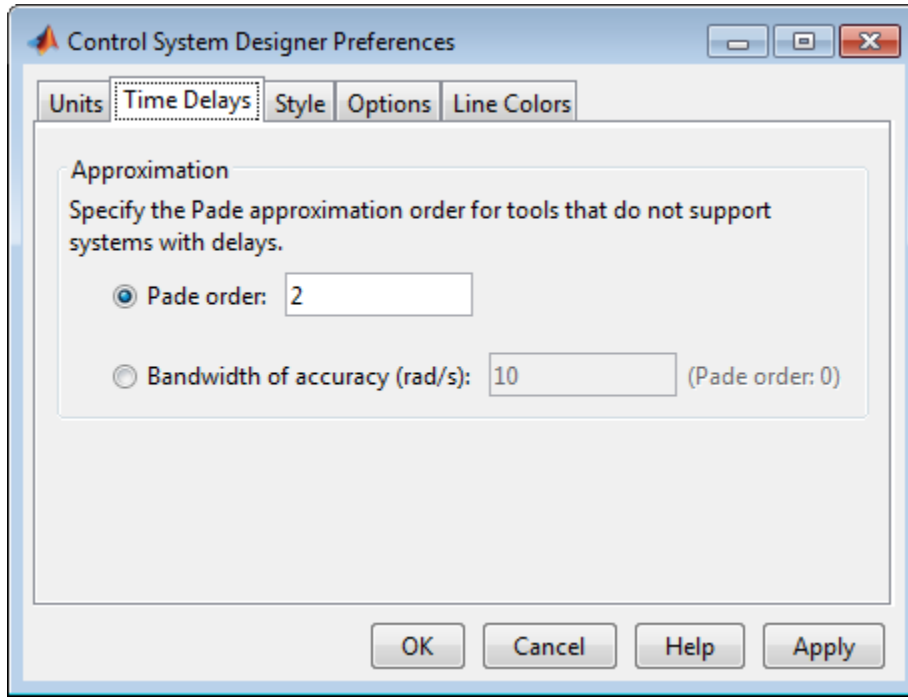
- Root locus plots
- Pole-zero maps
- PID automated tuning
- IMC automated tuning
- LQG automated tuning
- Loop shaping automated tuning

For these design methods and plots, the app automatically computes a Padé approximation for any exact delays in your model using a default Padé order.

To specify the default order, in **Control System Designer**, on the **Control System** tab, click **Preferences**.

In the Control System Designer Preferences dialog box, on the **Time Delays** tab, specify one of the following:

- **Pade order** — Specific Padé order.
- **Bandwidth of accuracy** — Highest frequency at which the approximated response matches the exact response. The app computes and displays the corresponding Padé order.



For more information on designing compensators using **Control System Designer**, see "Control System Designer Tuning Methods" on page 8-6.

See Also

`pade`

More About

- "Choose a Control Design Approach" on page 8-2
- "What Blocks Are Tunable?" on page 8-12

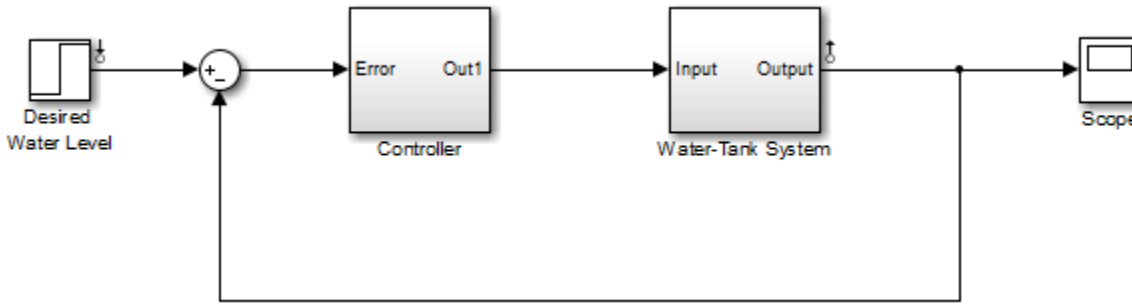
Design Compensator Using Automated PID Tuning and Graphical Bode Design

This example shows how to design a compensator for a Simulink model using automated PID tuning.

System Model

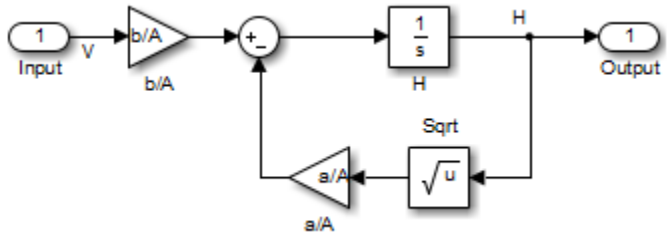
This example uses the `watertank_comp_design` Simulink model. To open the model, at the MATLAB command line, enter:

```
watertank_comp_design
```

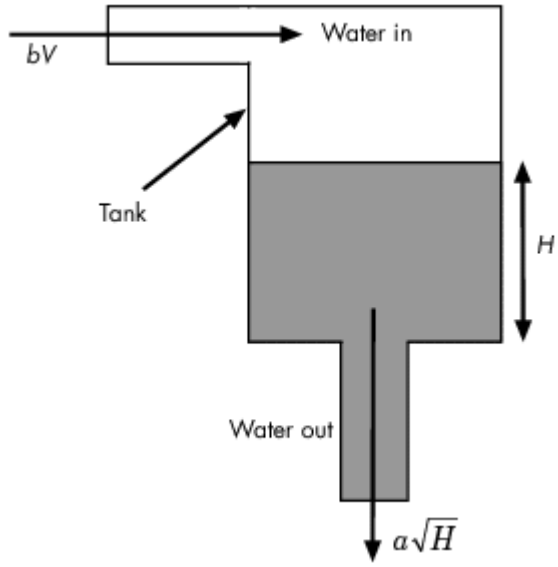


This model contains a Water-Tank System plant model and a PID controller in a single-loop feedback system.

To view the water tank model, double-click the Water-Tank System block.



This model represents the following water tank system:



where

- H is the height of water in the tank.
- Vol is the volume of water in the tank.
- V is the voltage applied to the pump.
- A is the cross-sectional area of the tank.
- b is a constant related to the flow rate into the tank.
- a is a constant related to the flow rate out of the tank.

Water enters the tank from the top at a rate proportional to the voltage applied to the pump. The water leaves through an opening in the tank base at a rate that is proportional to the square root of the water height in the tank. The presence of the square root in the water flow rate results in a nonlinear plant. Based on these flow rates, the rate of change of the tank volume is:

$$\frac{d}{dt}Vol = A\frac{dH}{dt} = bV - a\sqrt{H}$$

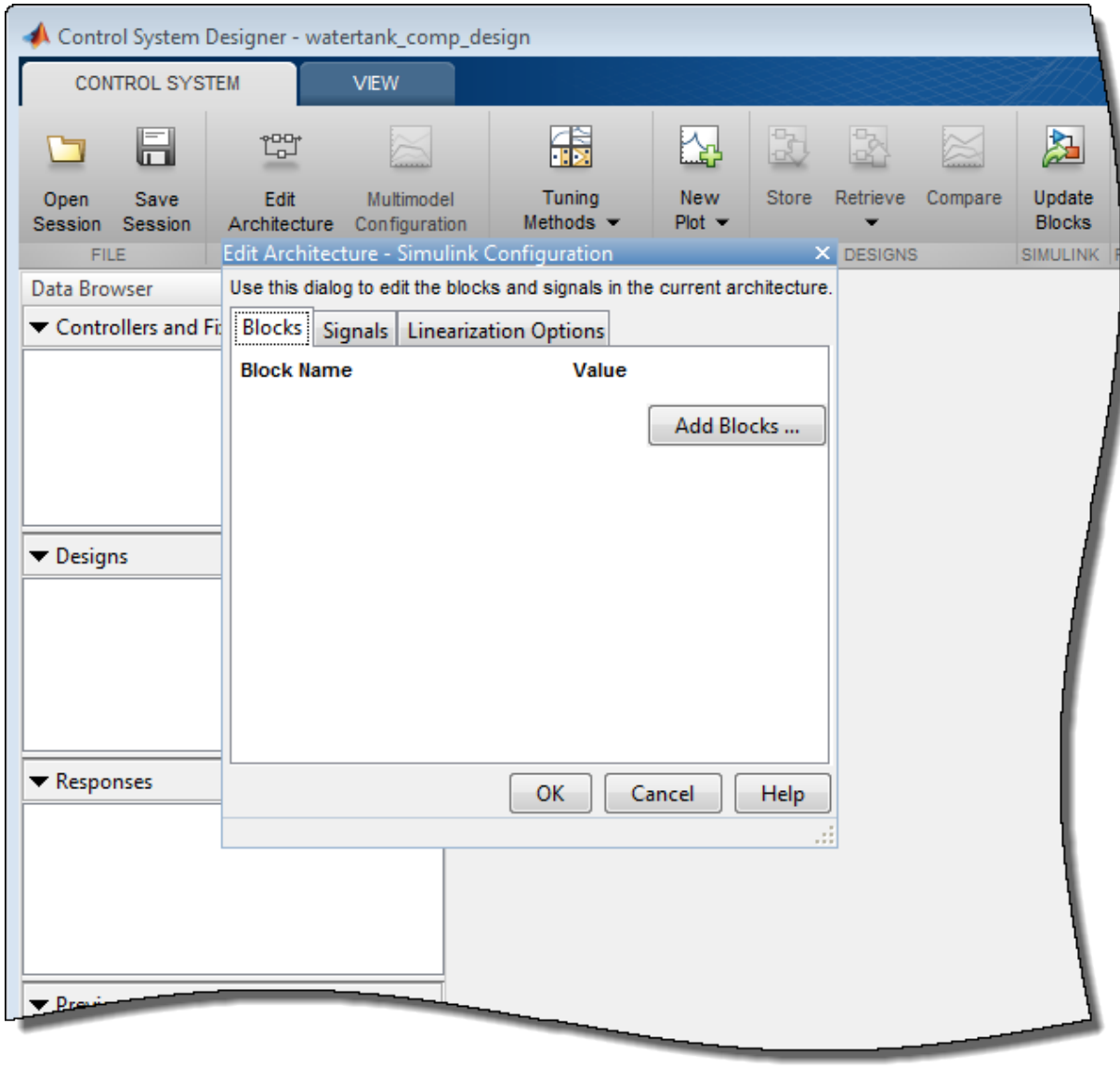
Design Requirements

Tune the PID controller to meet the following closed-loop step response design requirements:

- Overshoot less than 5%
- Rise time less than five seconds

Open Control System Designer

To open **Control System Designer**, in the Simulink model window, select **Analysis** > **Control Design** > **Control System Designer**.

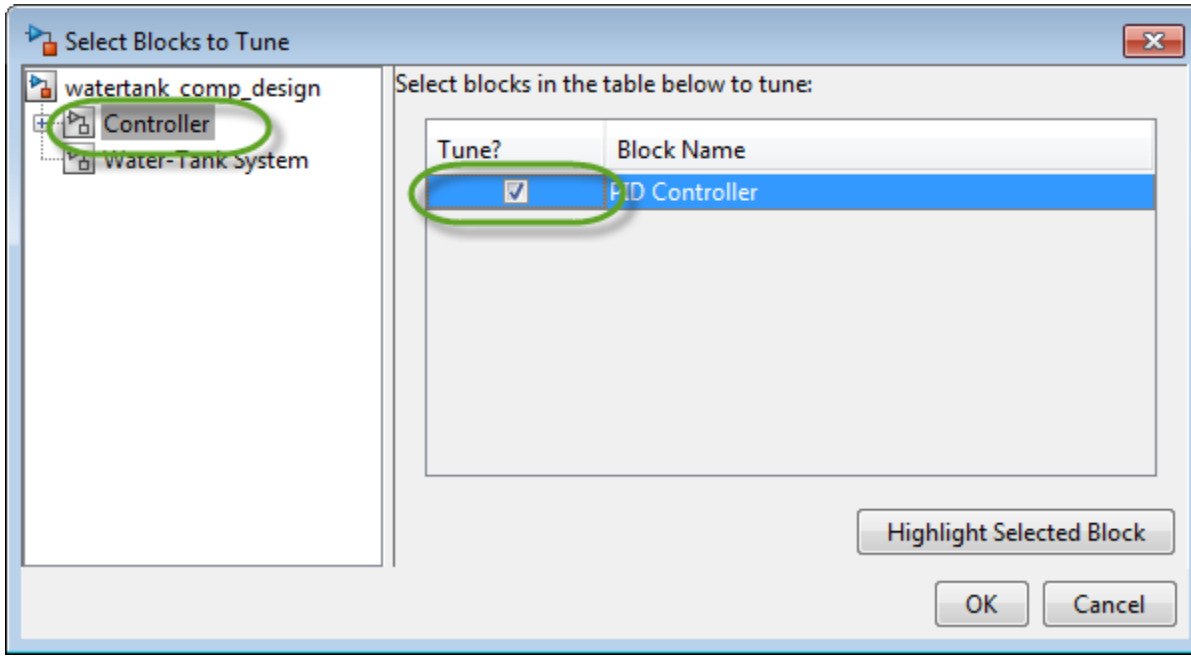


Control System Designer opens and automatically opens the Edit Architecture dialog box.

Specify Blocks to Tune

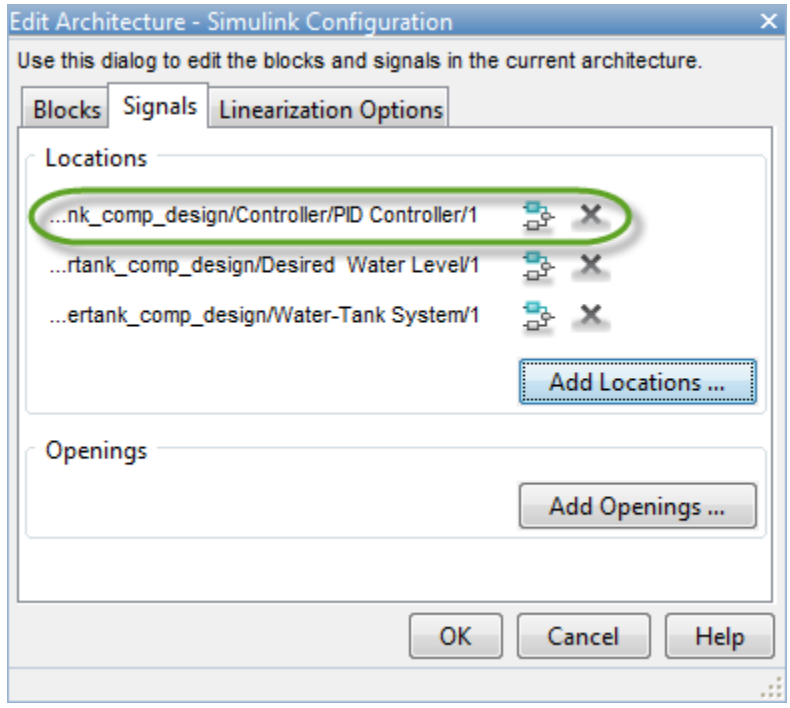
To specify the compensator to tune, in the Edit Architecture dialog box, click **Add Blocks**.

In the Select Blocks to Tune dialog box, in the left pane, click the Controller subsystem and, in the **Tune** column, check the box for the PID Controller.



Click **OK**.

In the Edit Architecture dialog box, the app adds the selected controller block to the list of blocks to tune on the **Blocks** tab. On the **Signals** tab, the app also adds the output of the PID Controller block to the list of analysis point **Locations**.



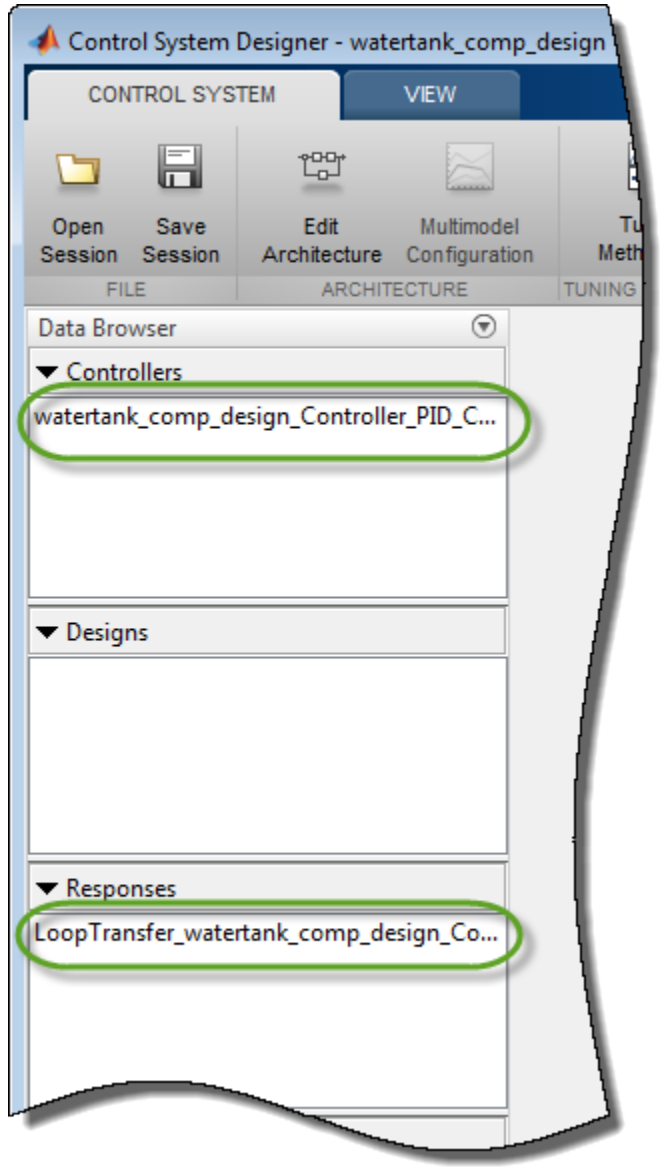
When **Control System Designer** opens, it adds any analysis points previously defined in the Simulink model to the **Locations** list. For the `watertank_comp_design`, there are two such signals.

- Desired Water Level block output — Reference signal for the closed-loop step response
- Water-Tank System block output — Output signal for the closed-loop step response

To linearize the Simulink model and set the control architecture, click **OK**.

By default, **Control System Designer** linearizes the plant model at the model initial conditions.

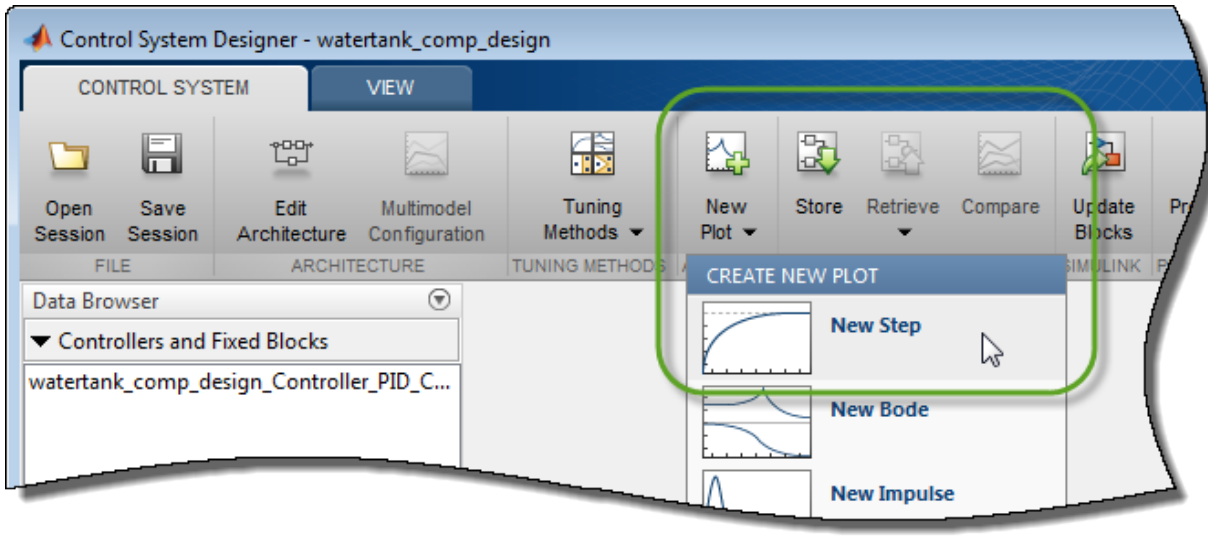
The app adds the PID controller to the **Data Browser**, in the **Controllers and Fixed Blocks** area. The app also computes the open-loop transfer function at the output of the PID Controller block, and adds this response to the **Data Browser**.


Control System Designer - watertank_comp_design
CONTROL SYSTEM
VIEW
Open Session
Save Session
Edit Architecture
Multimodel Configuration
FILE
ARCHITECTURE
Data Browser
Controllers
watertank_comp_design_Controller_PID_C...
Designs
Responses
LoopTransfer_watertank_comp_design_Co...

Plot Closed-Loop Step Response

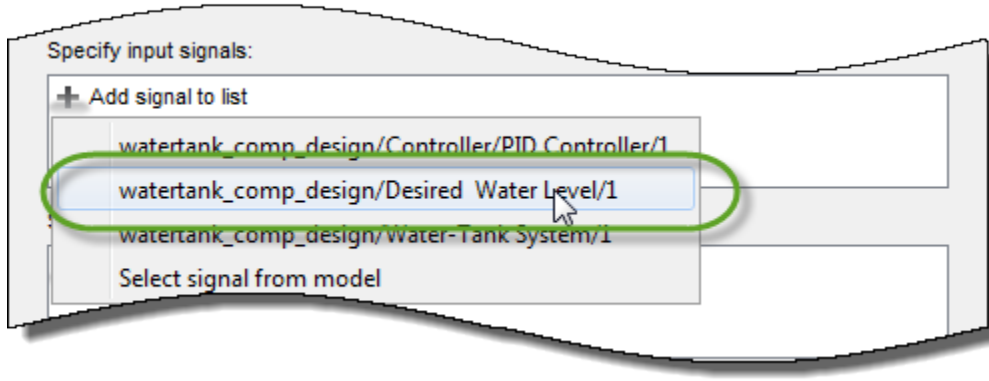
To analyze the controller design, create a closed-loop transfer function of the system, and plot its step response.

On the **Control System** tab, click **New Plot**, and select `New Step`.

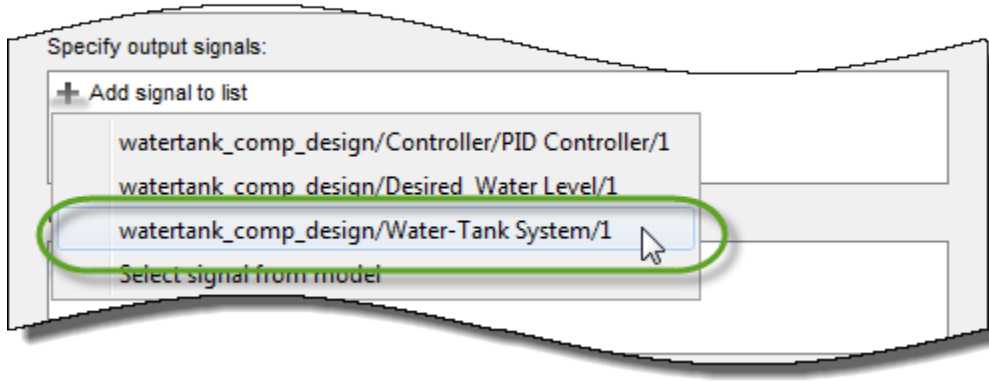


In the New Step to plot dialog box, in the **Select Response to Plot** drop-down list, select `New Input-Output Transfer Response`.

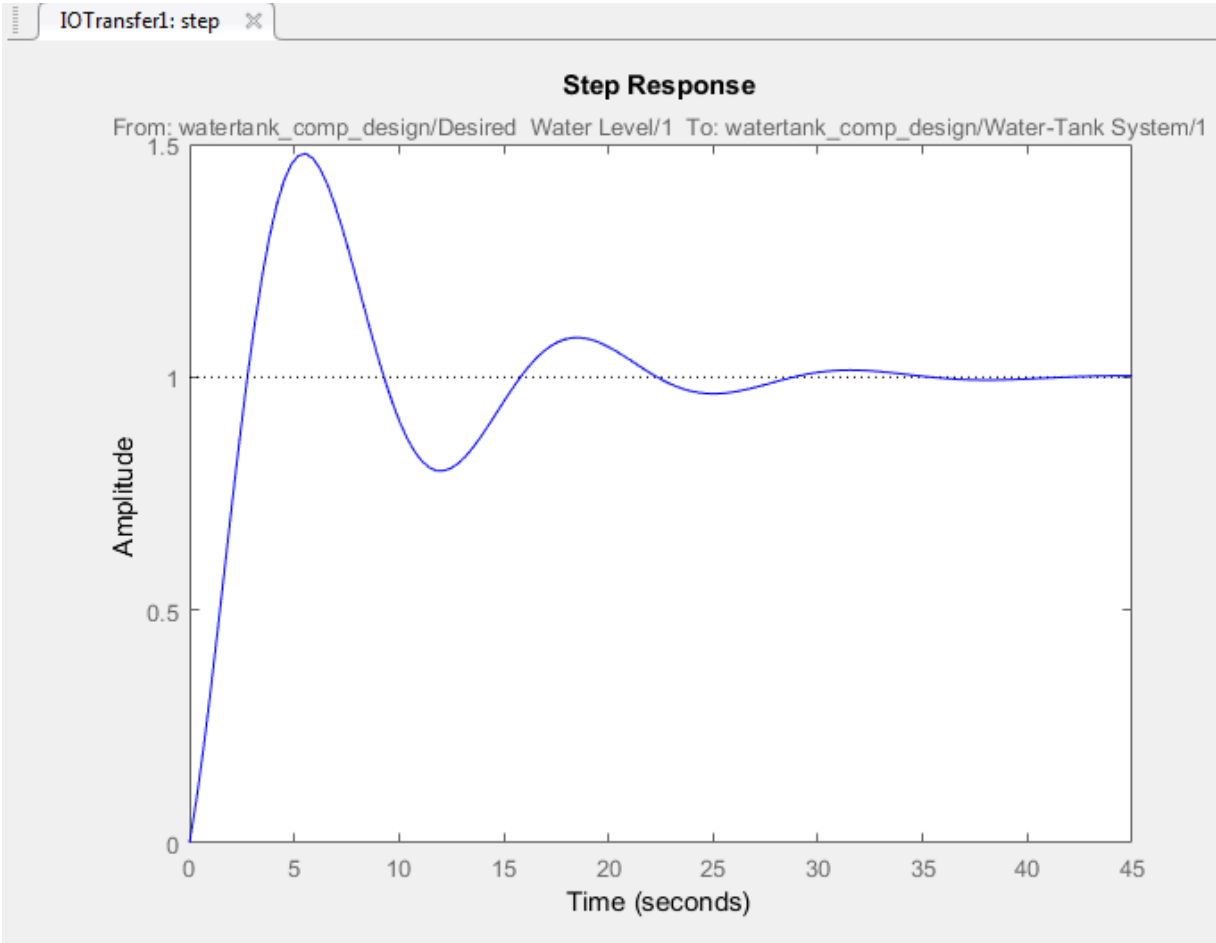
To add an input signal, in the **Specify input signals** area, click +. In the drop-down list, select the output of the Desired Water Level block.



To add an output signal, in the **Specify output signals** area, click +. In the drop-down list, select the output of the Water-Tank System block.

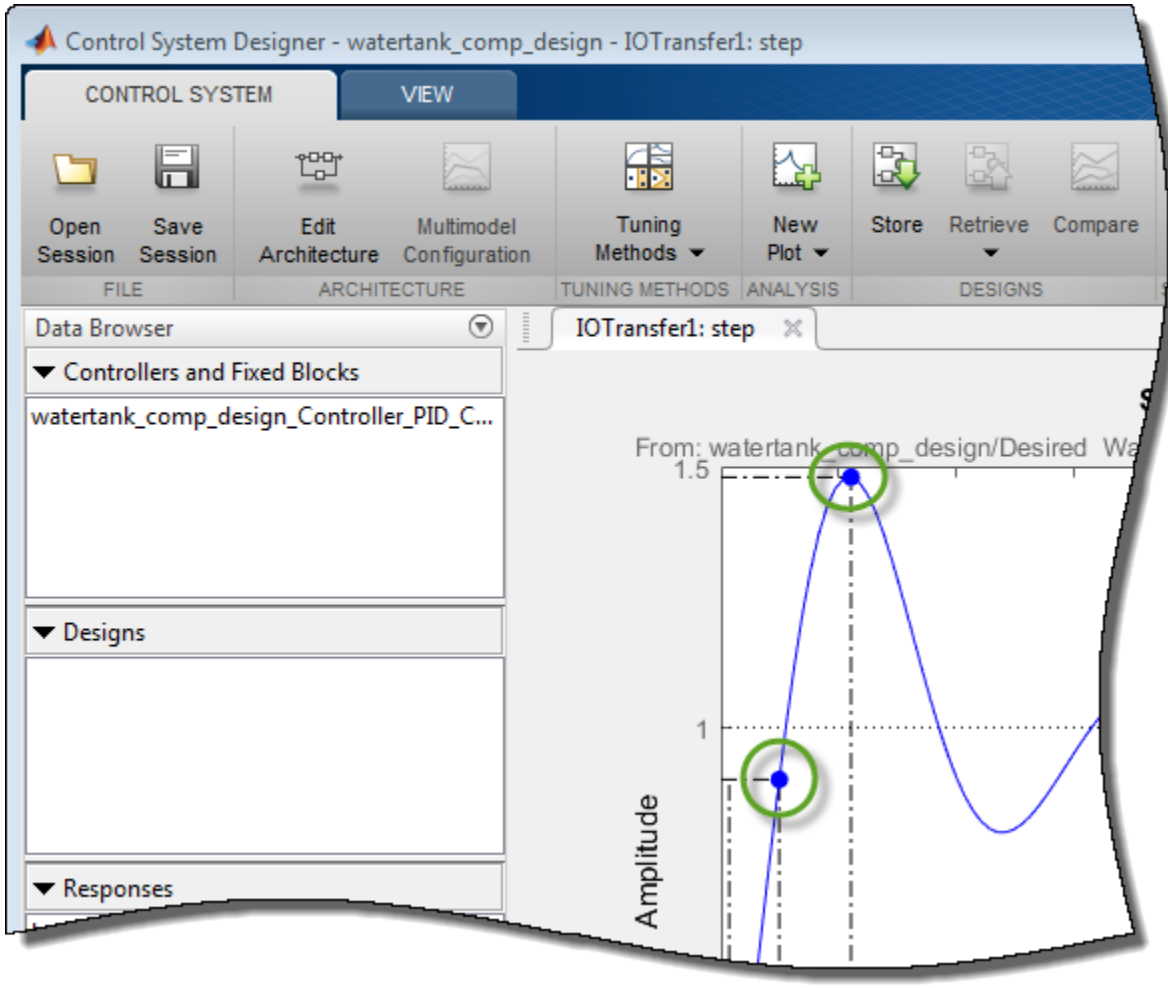


To create the closed-loop transfer function and plot the step response, click **Plot**.



To view the maximum overshoot on the response plot, right-click the plot area, and select **Characteristics > Peak Response**.

To view the rise time on the response plot, right-click the plot area, and select **Characteristics > Rise Time**.



Mouse-over the characteristic indicators to view their values. The current design has a:

- Maximum overshoot of 47.9%.
- Rise time of 2.13 seconds.

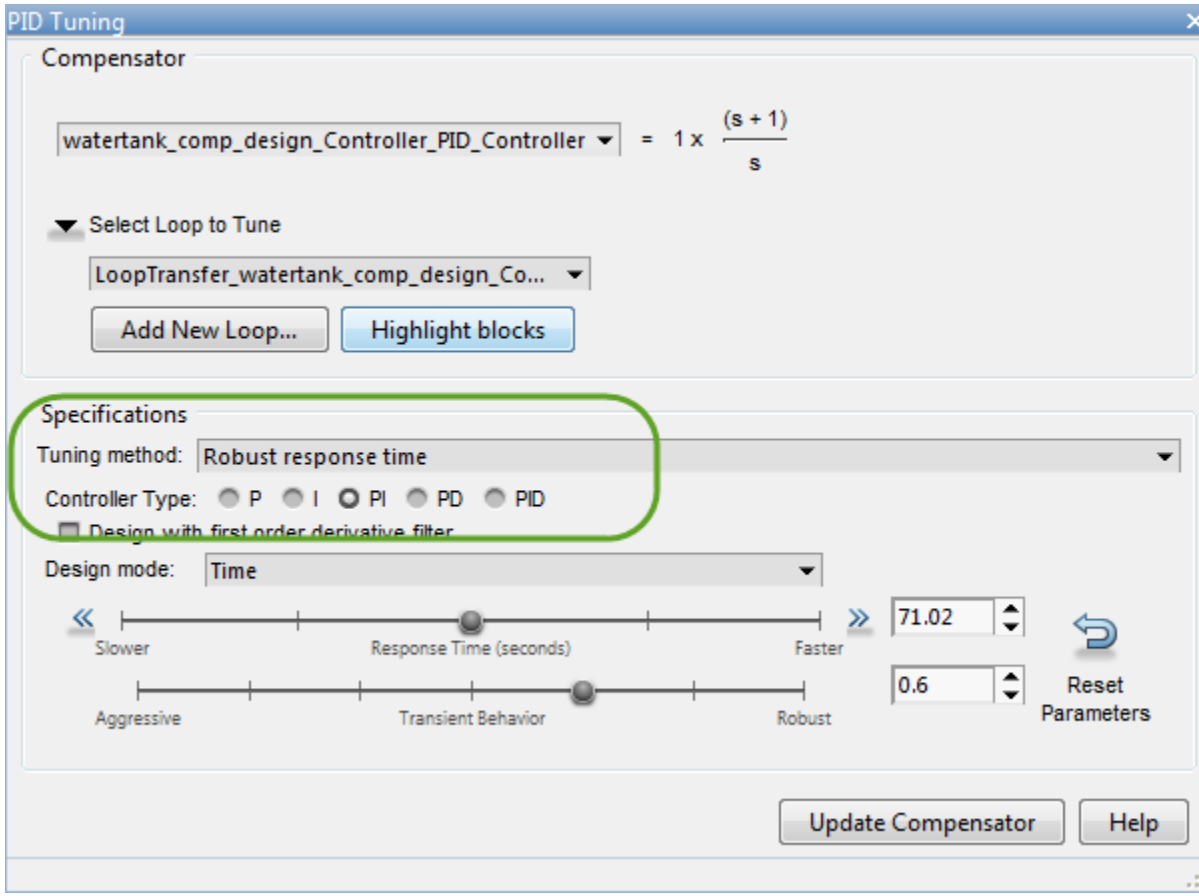
This response does not satisfy the 5% overshoot design requirement.

Tune Compensator Using Automated PID Tuning

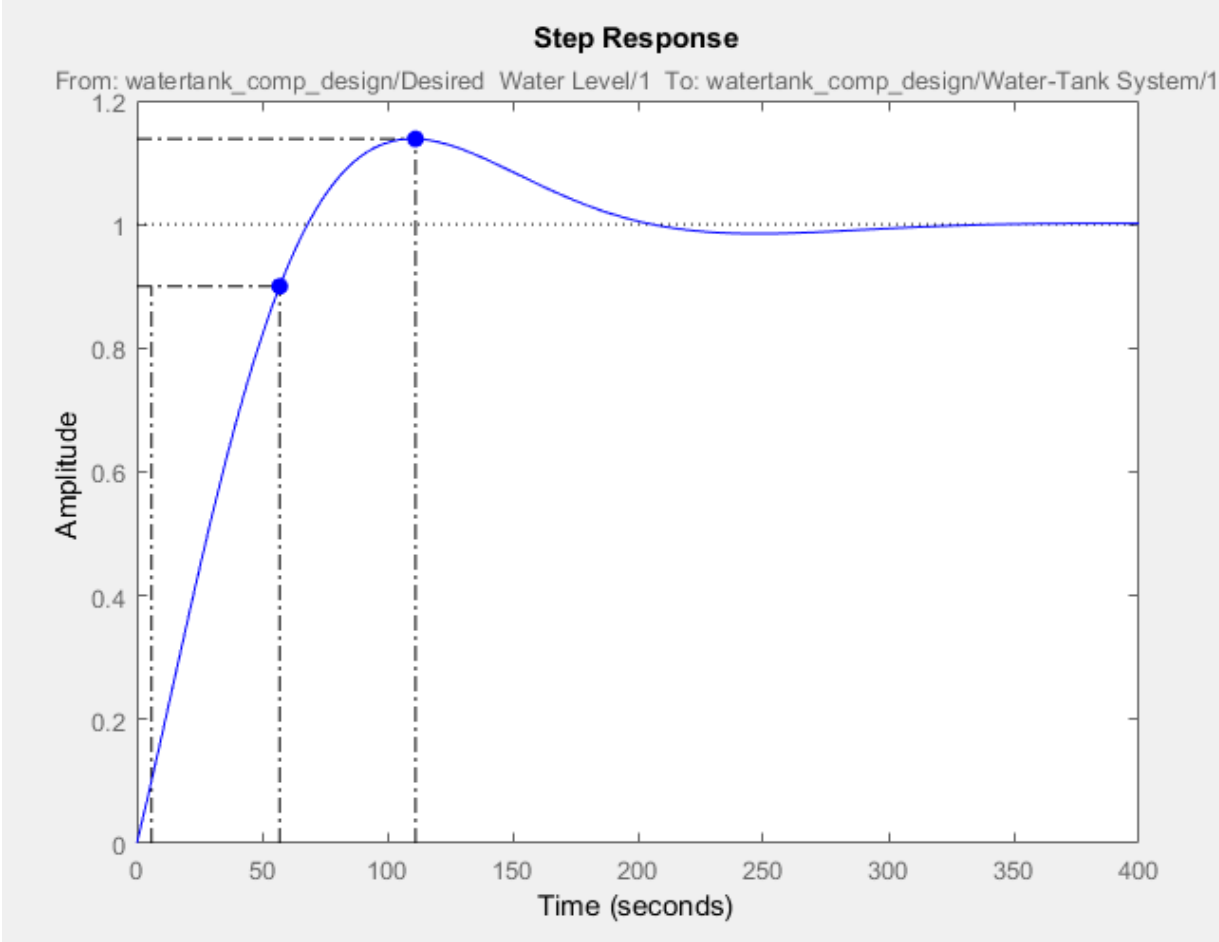
To tune the compensator using automated PID tuning, click **Tuning Methods**, and select `PID Tuning`.

In the PID Tuning dialog box, in the **Specifications** section, select the following options:

- **Tuning method** — `Robust response time`
- **Controller Type** — **PI**



Click **Update Compensator**. The app updates the closed-loop response for the new compensator settings and updates the step response plot.



To check the system performance, mouse over the response characteristic markers. The system response with the tuned compensator has a:

- Maximum overshoot of 13.8%.
- Rise time of 51.2 seconds.

This response exceeds the maximum allowed overshoot of 5%. The rise time is much slower than the required rise time of five seconds.

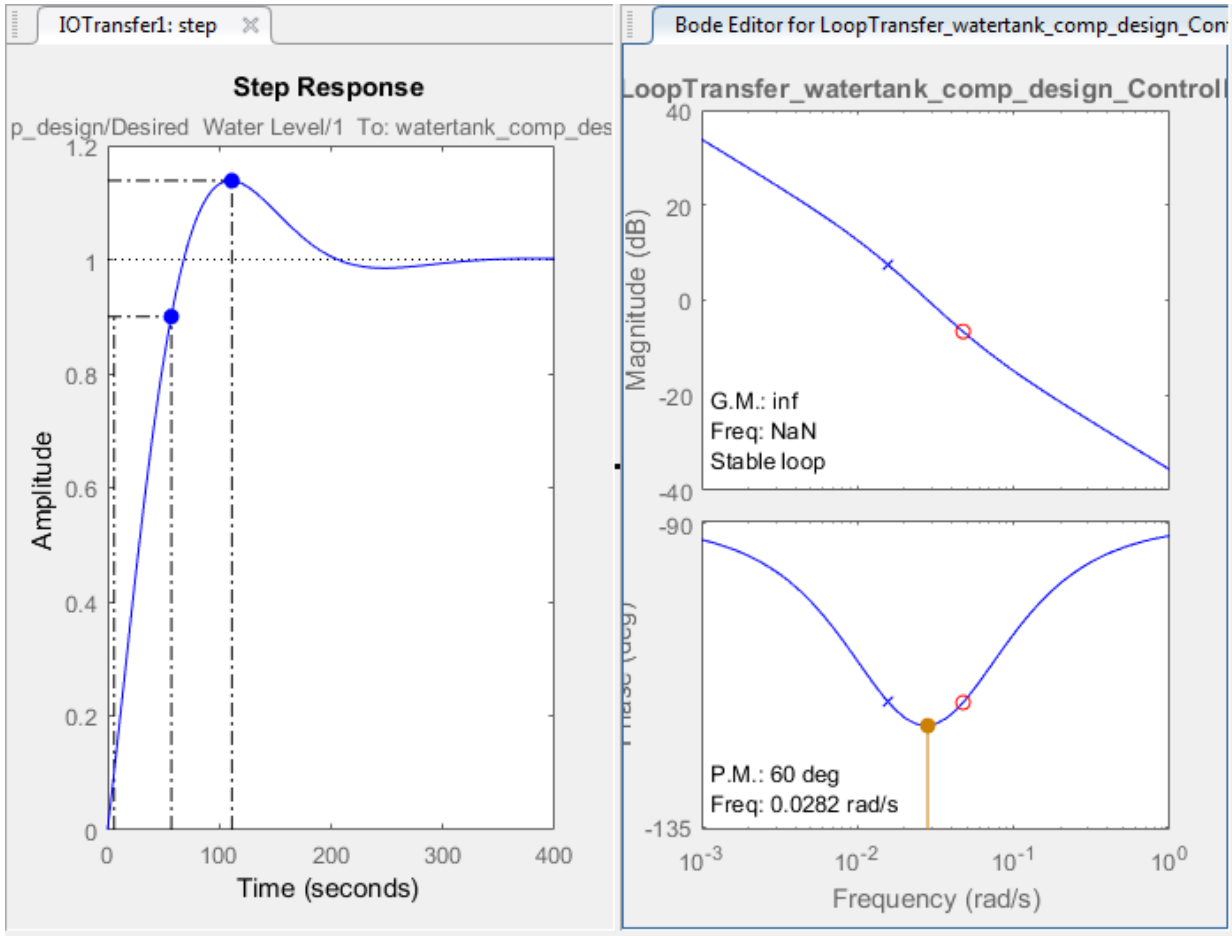
Tune Compensator Using Bode Graphical Tuning

To decrease the rise time, interactively increase the compensator gain using graphical Bode Tuning.

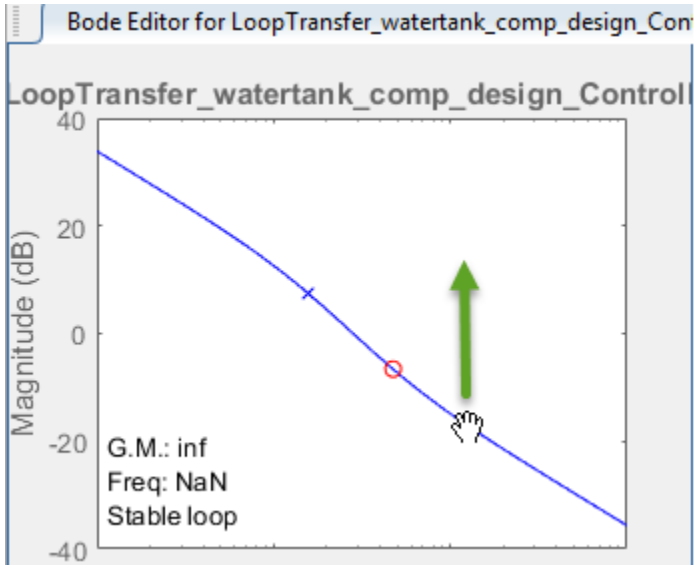
To open the open-loop Bode editor, click **Tuning Methods**, and select `Bode Editor`.

In the Select Response to Edit dialog box, the open-loop response at the output of the PID Controller block is already selected. To open the Bode editor for this response, click **Plot**.

To view the **Bode Editor** and **Step Response** plots side-by-side, on the **View** tab, click **Left/Right**.

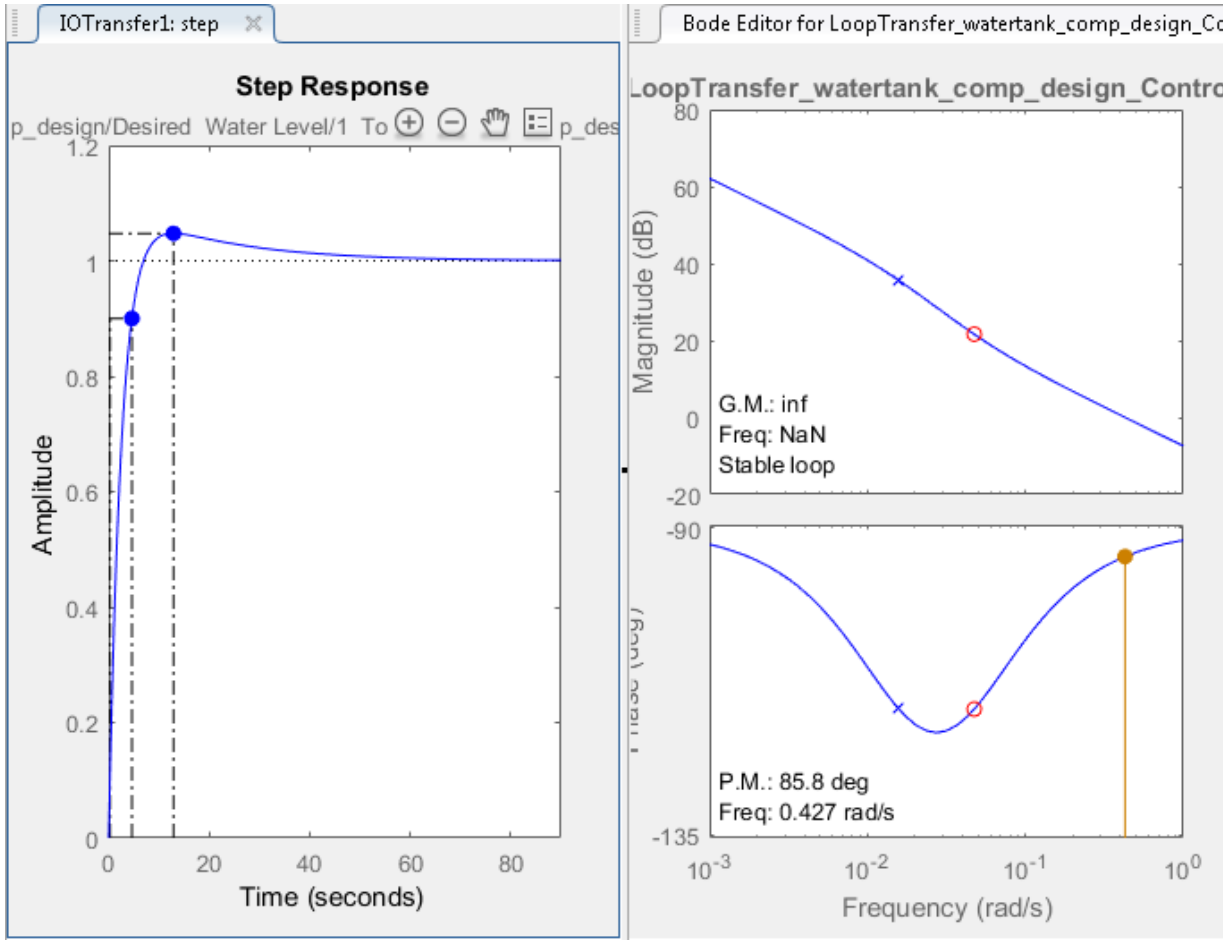


In the **Bode Editor** plot, drag the magnitude response up to increase the compensator gain. By increasing the gain, you increase the bandwidth and speed up the response.



As you drag the Bode response upward, the app automatically updates the compensator and the associated response plots. Also, when you release the plot, in the status bar, on the right side, the app displays the updated gain value.

Increase the compensator gain until the step response meets the design requirements. One potential solution is to set the gain to `1.7`.



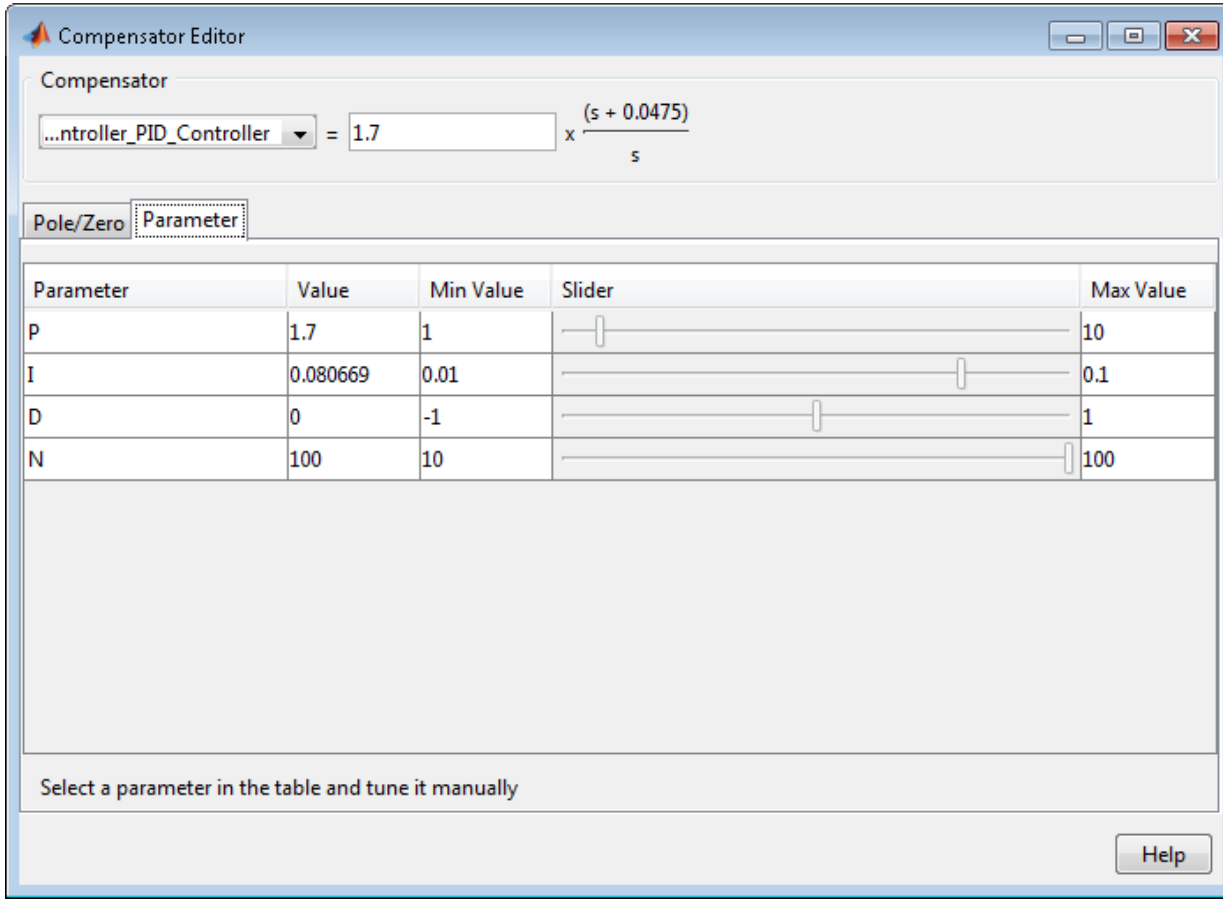
At this gain value, the closed loop response has a:

- Maximum overshoot of 4.74%.
- Rise time of 4.36 seconds.

Fine Tune Controller Using Compensator Editor

To tune the parameters of your compensator directly, use the compensator editor. In the **Bode Editor**, right-click the plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, on the **Parameter** tab, tune the PID controller gains. For more information on editing compensator parameters, see "Tuning Simulink Blocks in the Compensator Editor" on page 8-83.



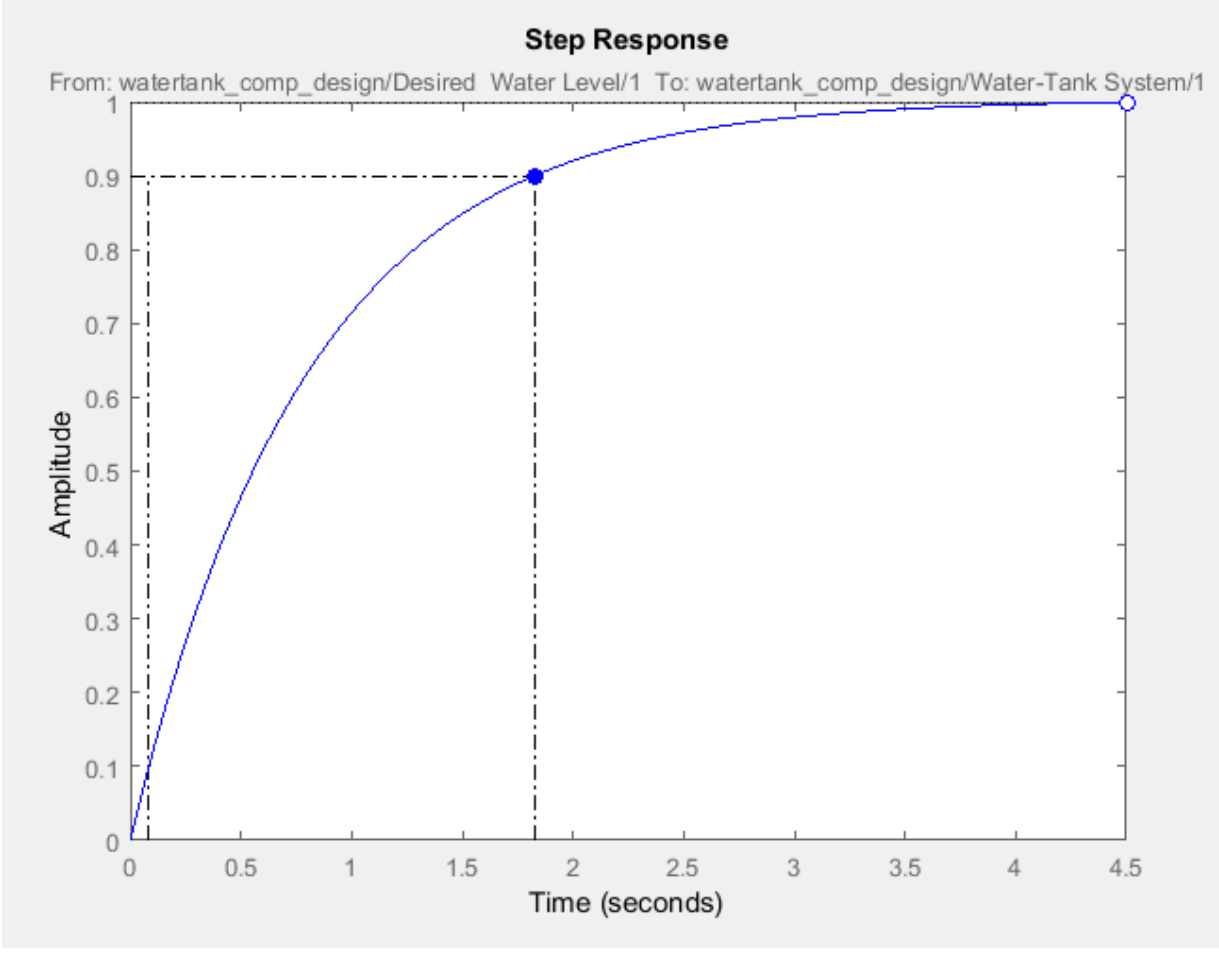
While the tuned compensator meets the design requirements, the settling time is over 30 seconds. To improve the settling time, adjust the **P** and **I** parameters of the controller manually.

For example, set the compensator parameters to:

- **P** = `4`
- **I** = `0.1`

This compensator produces a closed-loop response with a:

- Maximum overshoot of 0.206%.
- Rise time of 1.74 seconds.
- Settling time of around three seconds.



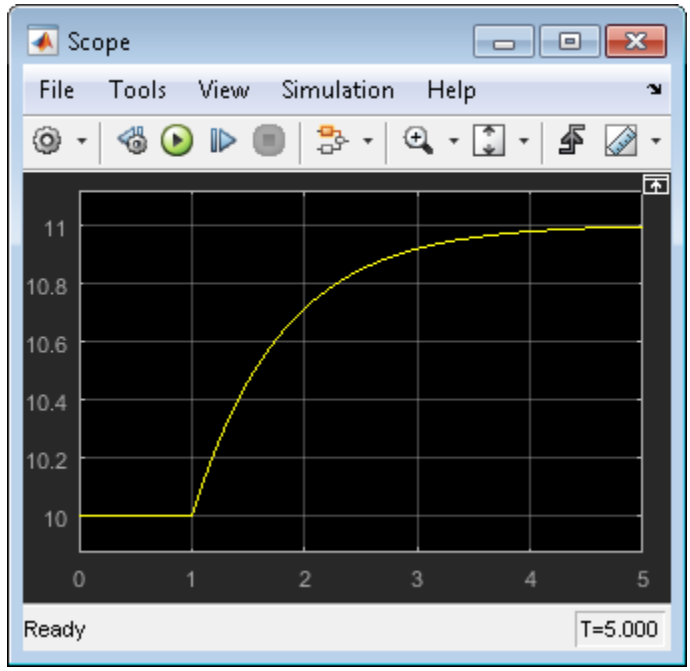
Simulate Closed-Loop System in Simulink

Validate your compensator design by simulating the nonlinear Simulink model with the tuned controller parameters.

To write the tuned compensator parameters to the PID Controller block, in **Control System Designer**, on the **Control System** tab, click **Update Blocks**.

In the Simulink model window, run the simulation.

To view the closed-loop simulation output, double-click theScope block.



The closed-loop response of the nonlinear system satisfies the design requirements with a rise time of less than five seconds and minimal overshoot.

See Also

More About

- "Control System Designer Tuning Methods" on page 8-6
- "Update Simulink Model and Validate Design" on page 8-51

Analyze Designs Using Response Plots

This example shows how to analyze your control system designs using the plotting tools in **Control System Designer**. There are two types of **Control System Designer** plots:

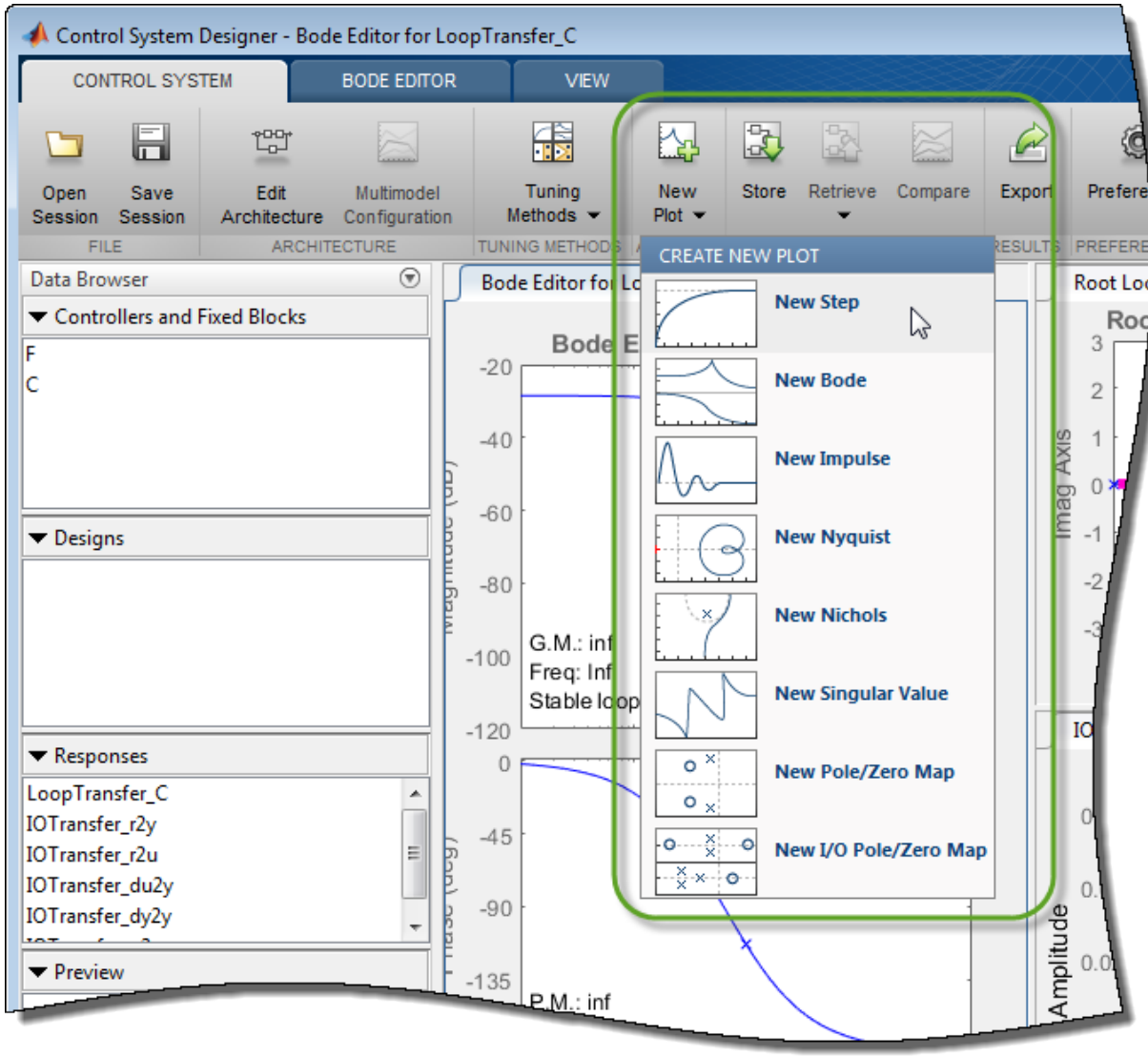
- Analysis plots — Use these plots to visualize your system performance and display response characteristics.
- Editor plots — Use these plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

In this section...

Analysis Plots

Use analysis plots to visualize your system performance and display response characteristics. You can also use these plots to compare multiple control system designs. For more information, see “Compare Performance of Multiple Designs” on page 8-46.

To create a new analysis plot, in **Control System Designer**, on the **Control System** tab, click **New Plot**, and select the type of plot to add.



In the new plot dialog box, specify an existing or new response to plot.

Note Using analysis plots, you can compare the performance of multiple designs stored in the **Data Browser**. For more information, see "Compare Performance of Multiple Designs" on page 8-46.

Plot Existing Response

To plot an existing response, in the **Select Response to Plot** drop-down list, select an existing response from the **Data Browser**. The details for the selected response are displayed in the text box.

To plot the selected response, click **Plot**.

Plot New Response

To plot a new response, specify the following:

- **Select Response to Plot** — Select the type of response to create.
  - `New Input-Output Transfer Response` — Create a transfer function response for specified input signals, output signals, and loop openings.
  - `New Open-Loop Response` — Create an open-loop transfer function response at a specified location with specified loop openings.
  - `New Sensitivity Transfer Response` — Create a sensitivity response at a specified location with specified loop openings.
- **Response Name** — Enter a name in the text box.
- Signal selection boxes — Specify signals as inputs, outputs, or loop openings by clicking  . If you open **Control System Designer** from:
  - MATLAB — Select a signal using the **Architecture** block diagram for reference.
  - Simulink — Select an existing signal from the current control system architecture, or add a signal from the Simulink model.

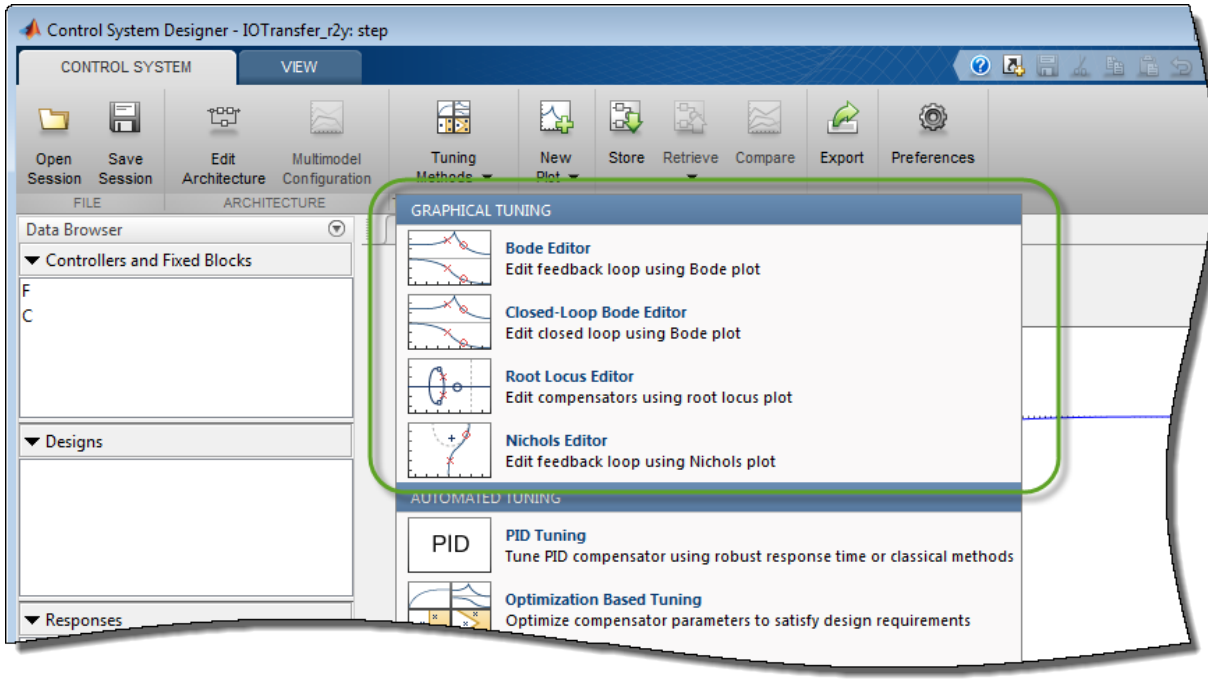
  Use  ,  , and  to reorder and delete signals.

To add the specified response to the **Data Browser** and create the selected plot, click **Plot**.

Editor Plots

Use editor plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

To create a new editor plot, in **Control System Designer**, on the **Control System** tab, click **Tuning Methods**, and select one of the **Graphical Tuning** methods.



For examples of graphical tuning using editor plots, see:

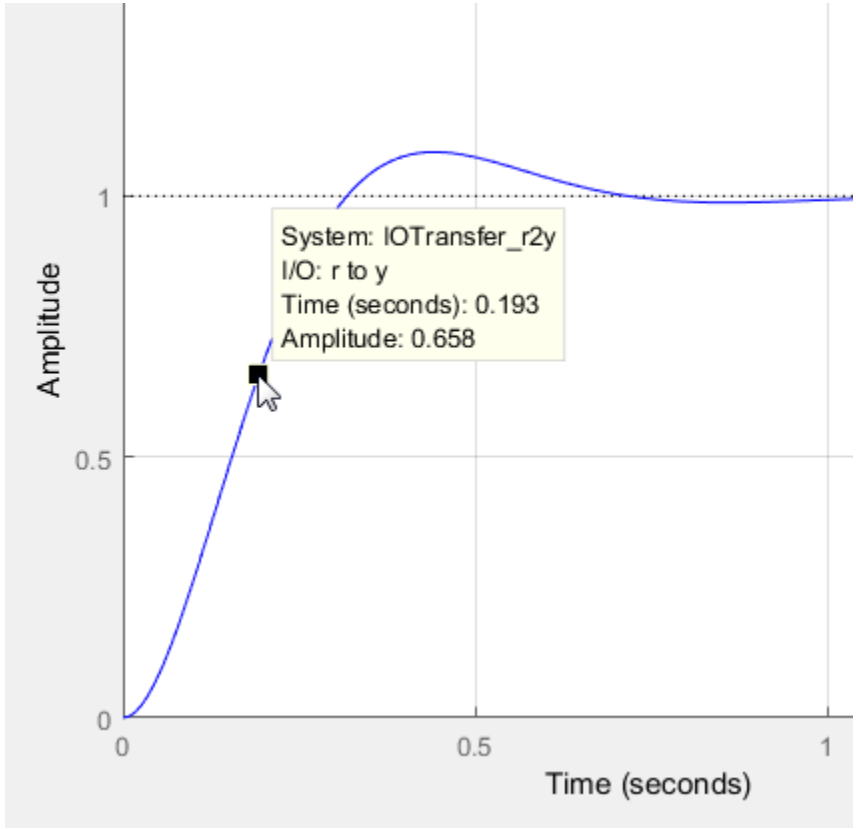
- “Bode Diagram Design” (Control System Toolbox)
- “Root Locus Design” (Control System Toolbox)
- “Nichols Plot Design” (Control System Toolbox)

For more information on interactively editing compensator dynamics, see “Edit Compensator Dynamics” (Control System Toolbox).

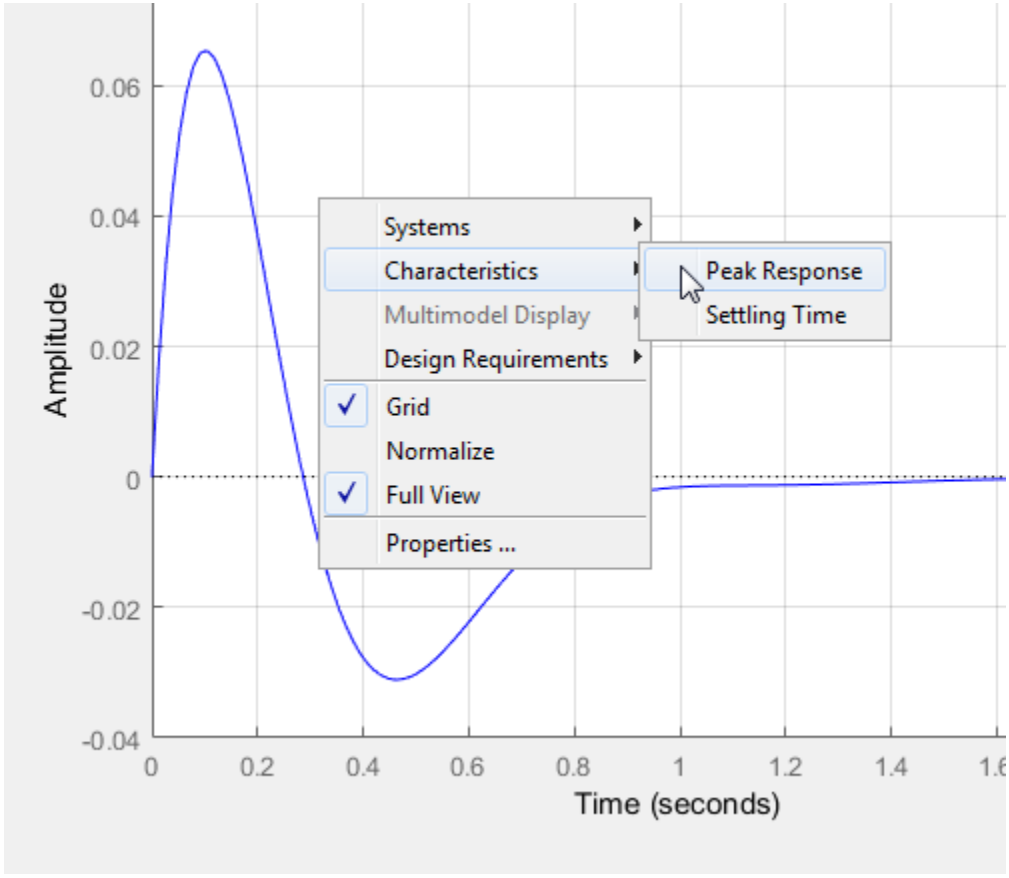
Plot Characteristics

On any analysis plot in **Control System Designer**:

- To see response information and data values, click a line on the plot.

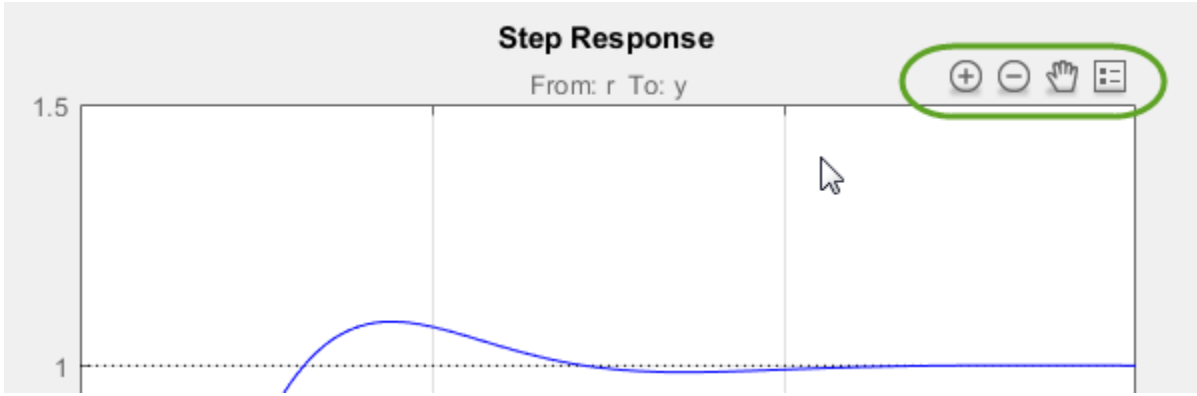


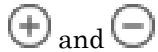
- To view system characteristics, right-click anywhere on the plot, as described in "Frequency-Domain Characteristics on Response Plots" (Control System Toolbox).

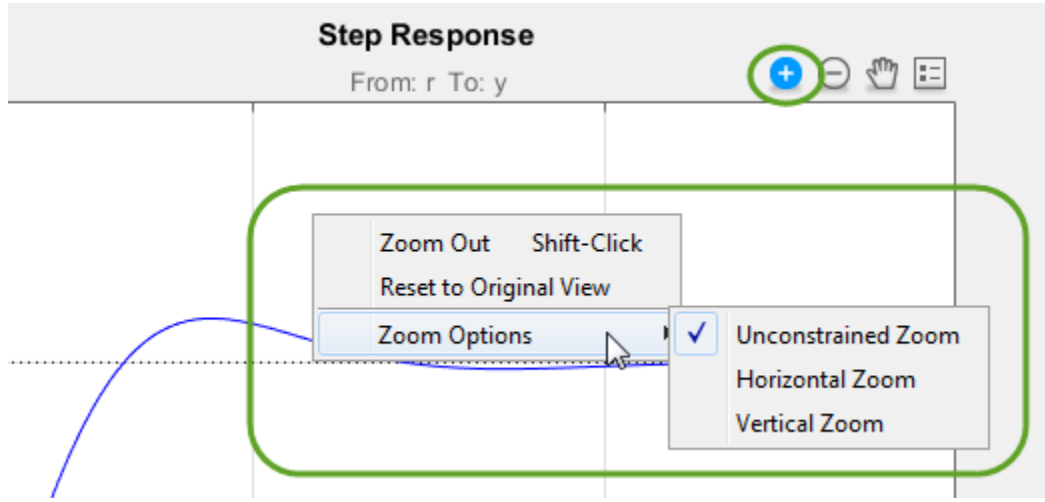


Plot Tools

Mouse over any analysis plot to access plot tools at the upper right corner of the plot.

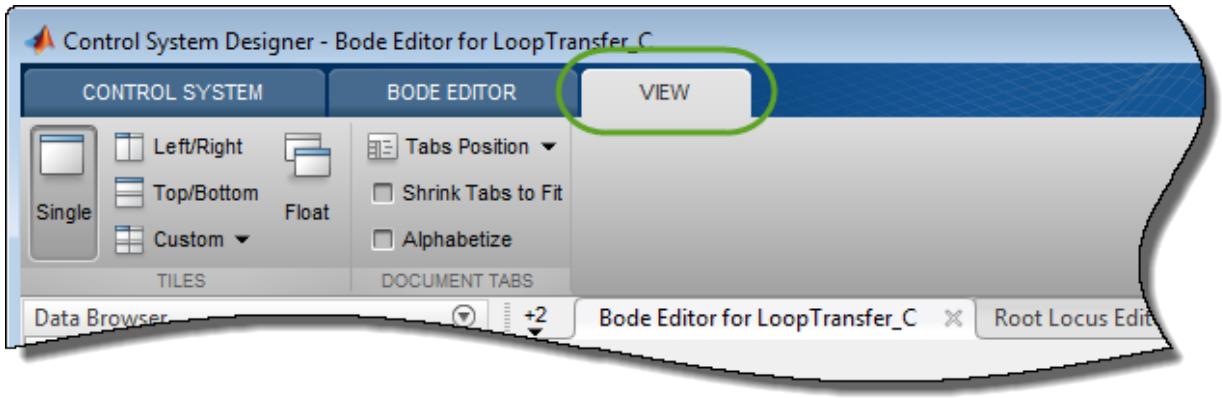


-  and — Zoom in and zoom out. Click to activate, and drag the cursor over the region to zoom. The zoom icon turns dark when zoom is active. Right-click while zoom is active to access additional zoom options. Click the icon again to deactivate.



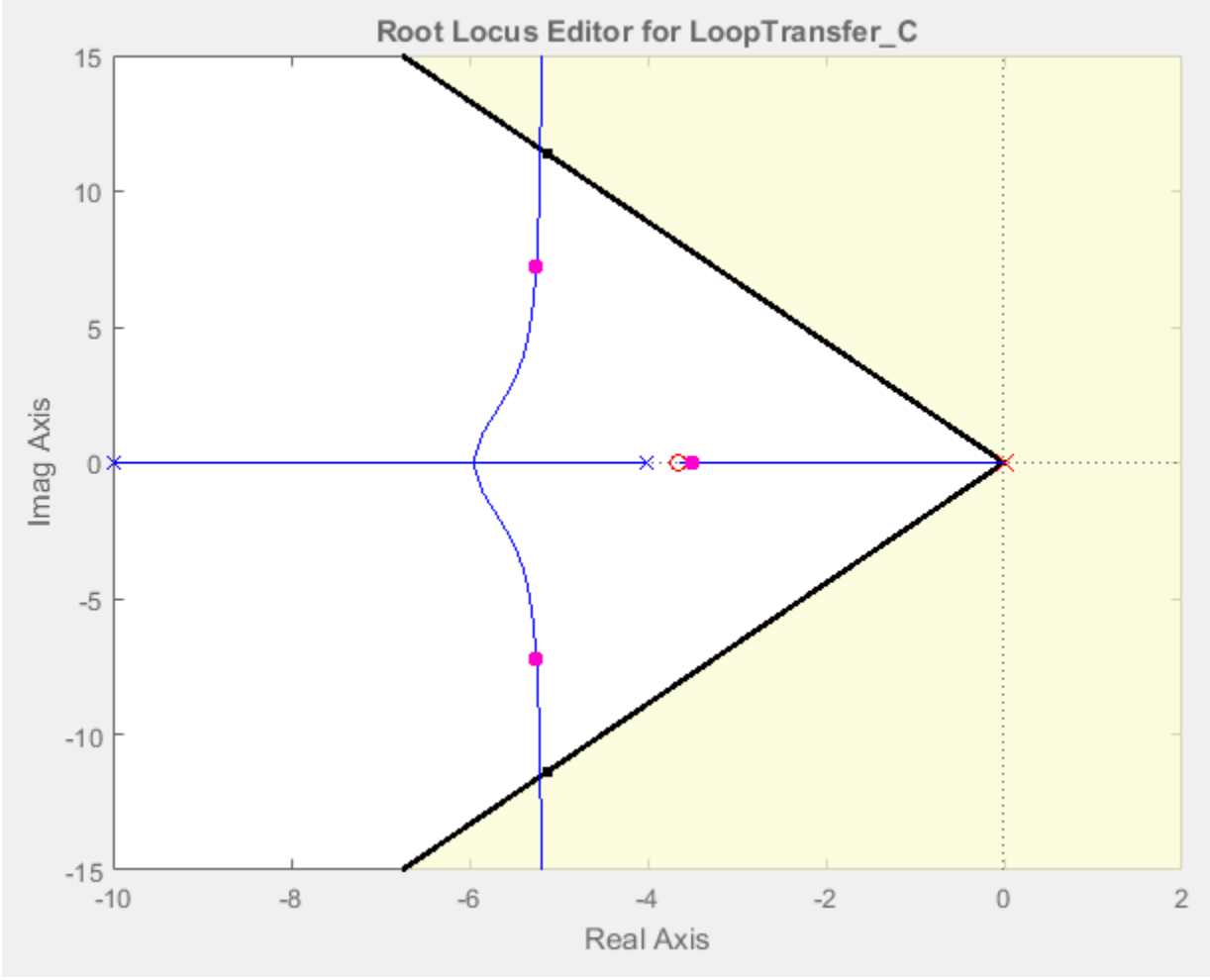
-  — Pan. Click to activate, and drag the cursor across the plot area to pan. The pan icon turns dark when pan is active. Right-click while pan is active to access additional pan options. Click the icon again to deactivate.
-  — Legend. By default, the plot legend is inactive. To toggle the legend on and off, click this icon. To move the legend, drag it to a new location on the plot.

To change the way plots are tiled or sorted, use the options on the **View** tab.



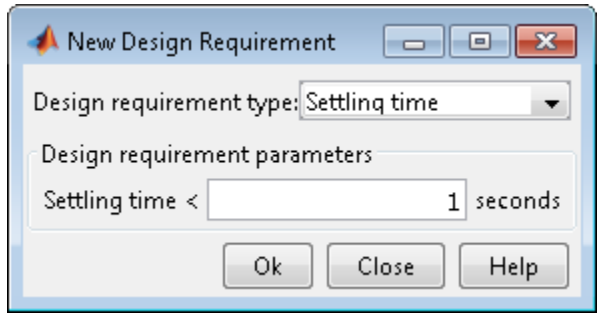
Design Requirements

You can add graphical representations of design requirements to any editor or analysis plots. These requirements define shaded exclusion regions in the plot area.



Use these regions as guidelines when analyzing and tuning your compensator designs. To meet a design requirement, your response plots must remain outside of the corresponding shaded area.

To add design requirements to a plot, right-click anywhere on the plot and select **Design Requirements > New**.



In the New Design Requirement dialog box, specify the **Design requirement type**, and define the **Design requirement parameters**. Each type of design requirement has a different set of parameters to configure. For more information on adding design requirements to analysis and editor plots, see “Design Requirements” (Control System Toolbox).

See Also

More About

- “Control System Designer Tuning Methods” on page 8-6
- “Compare Performance of Multiple Designs” on page 8-46
- “Design Requirements” (Control System Toolbox)

Compare Performance of Multiple Designs

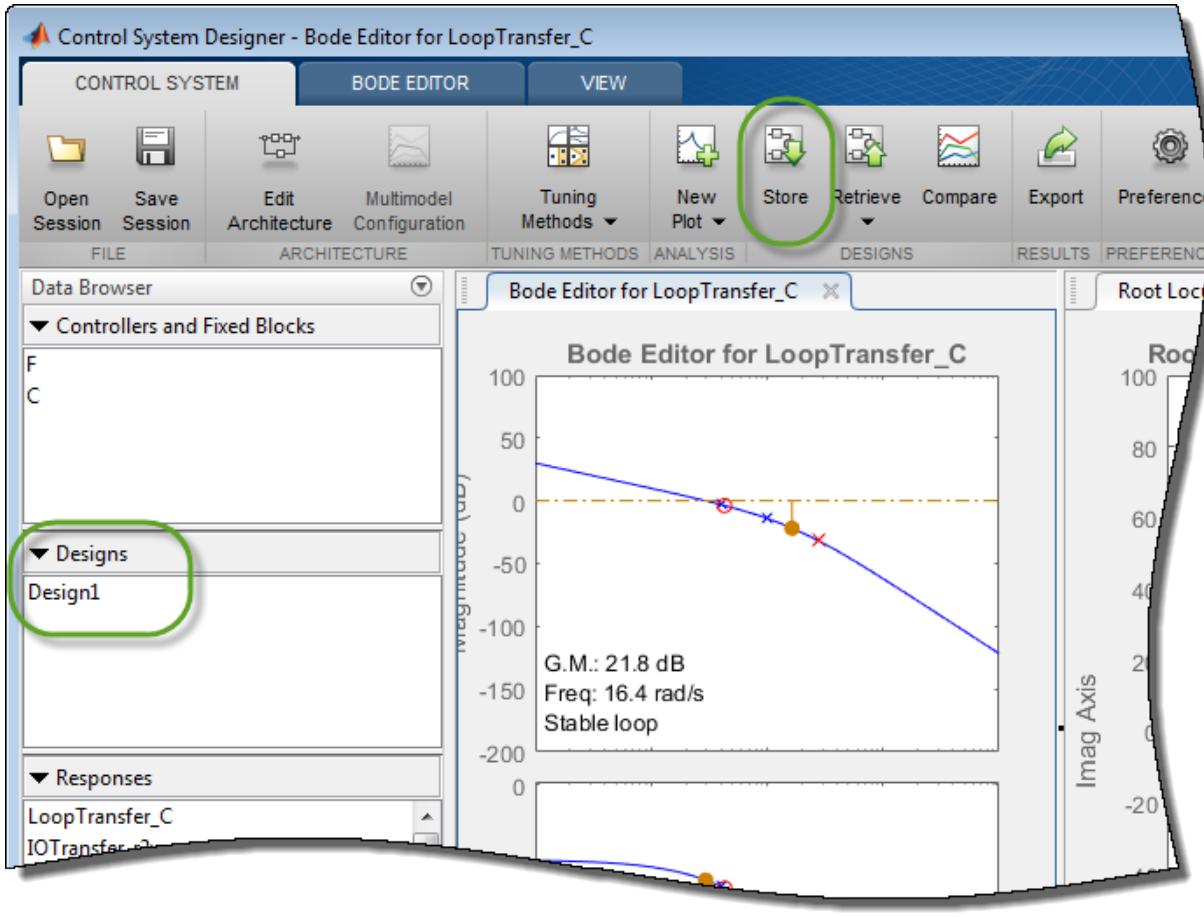
This example shows how to compare the performance of two different control system designs. Such comparison is useful, for example, to see the effects of different tuning methods or compensator structures.

Store First Design

In this example, the first design is the compensator tuned graphically in “Bode Diagram Design” (Control System Toolbox).

After tuning the compensator with this first tuning method, store the design in **Control System Designer**.

On the **Control System** tab, in the **Designs** section, click  **Store**. The stored design appears in the **Data Browser** in the **Designs** area.



To rename the stored design, in the **Data Browser**, double-click the design, and specify a new name.

Compute New Design

On the **Control System** tab, tune the compensator using a different tuning method.

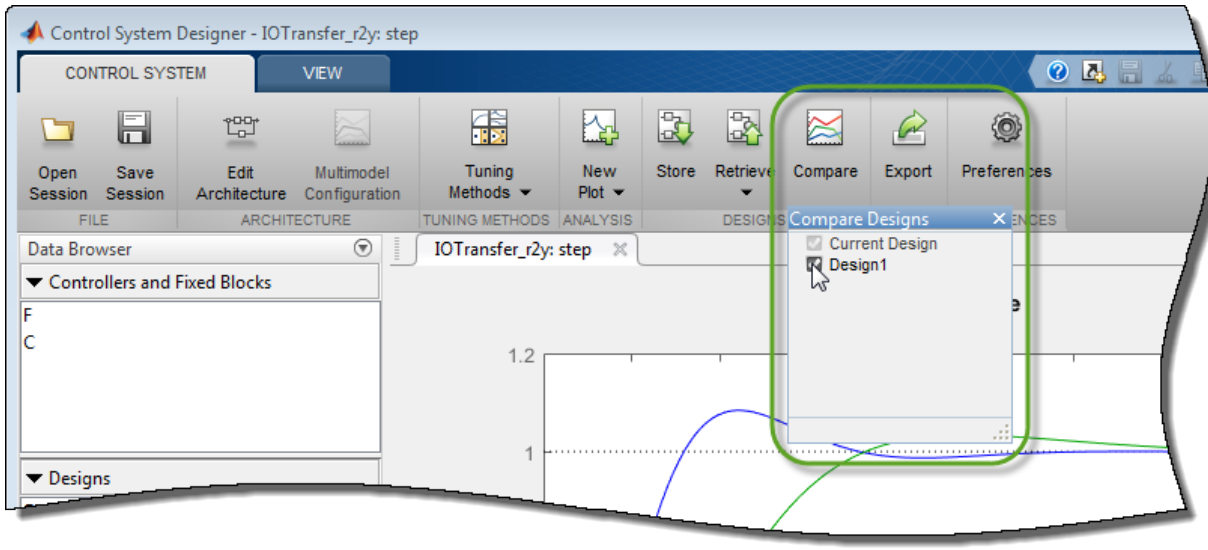
Under **Tuning Methods**, select `PID Tuning`.

To design a controller with the default `Robust response time` specifications, in the PID Tuning dialog box, click **Update Compensator**.

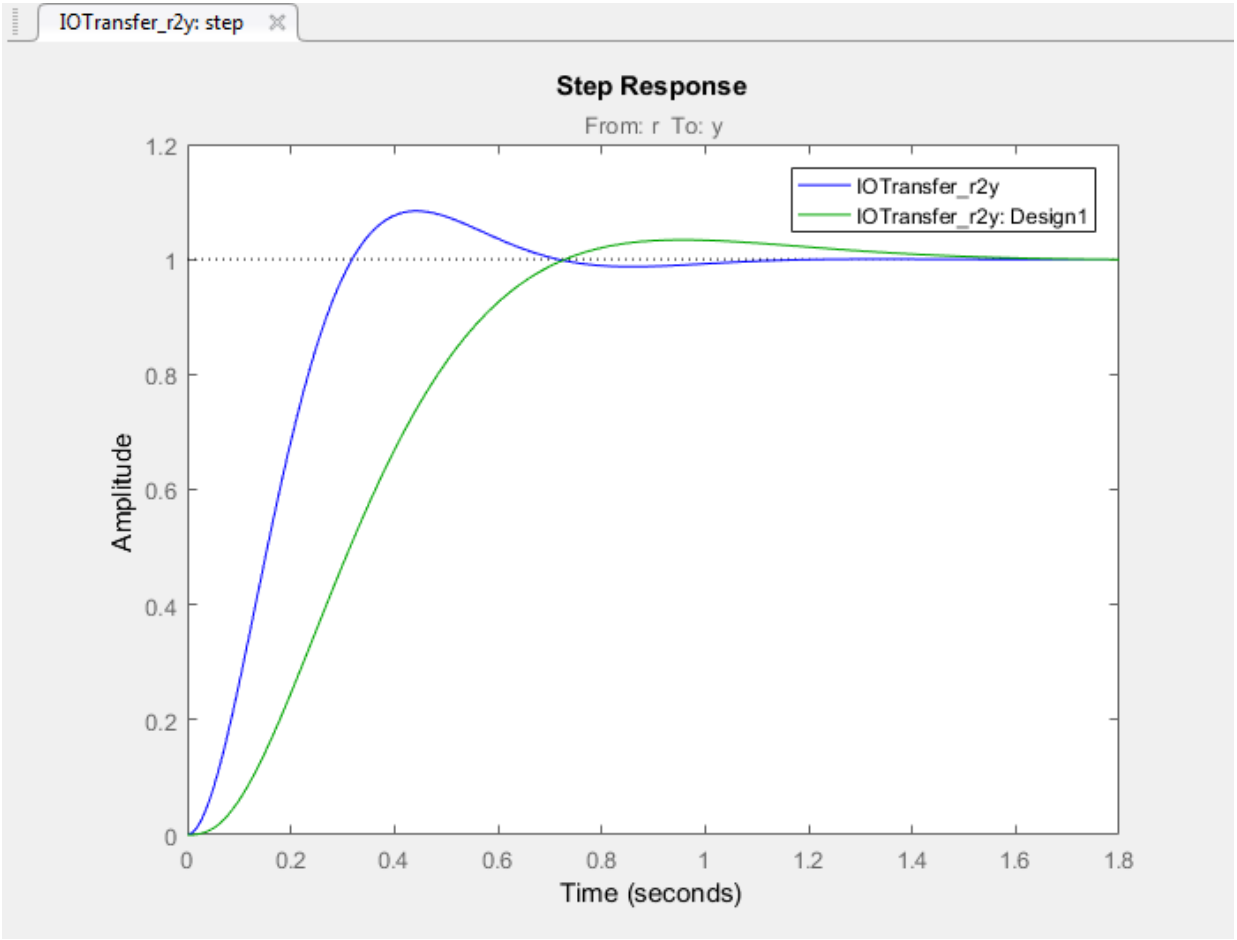
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design.

On the **Control System** tab, click **Compare**.



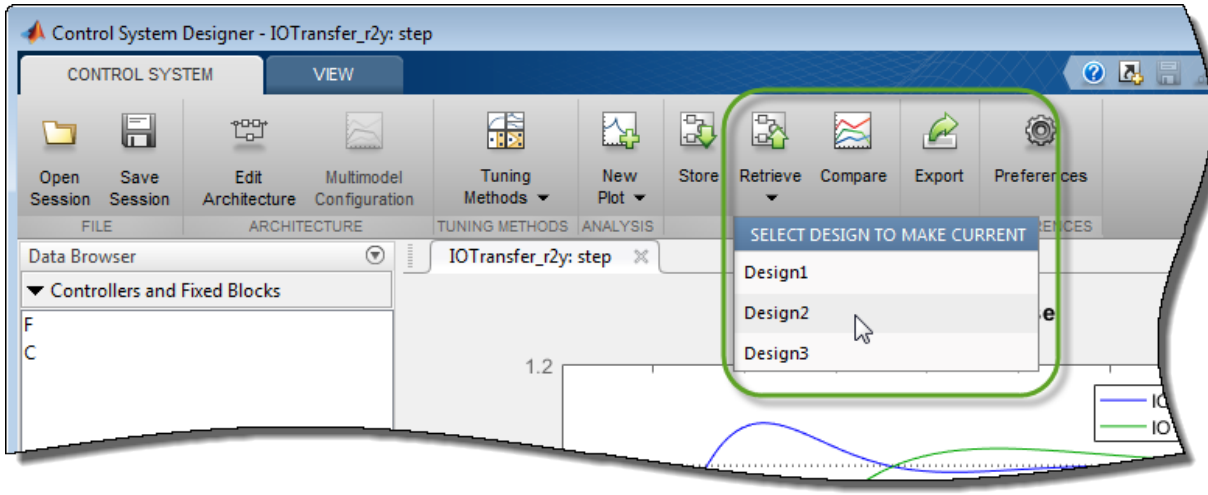
In the **Compare Designs** dialog box, the current design is checked by default. To compare a design with the current design, check the corresponding box. All analysis plots update to reflect the checked designs. The blue trace corresponds to the current design. Refer to the plot legend to identify the responses for other designs.



Restore Previously Saved Design

Under some conditions, it is useful to restore a previously stored design. For example, when designing a compensator for a Simulink model, you can write the current compensator values to the model (see “Update Simulink Model and Validate Design” on page 8-51). To test a stored compensator in your model, first restore the stored design as the current design.

To do so, in **Control System Designer**, click **Retrieve**. Select the stored design that you want to make current.



Note The retrieved design overwrites the current design. If necessary, store the current design before retrieving a previously stored design.

See Also

More About

- "Analyze Designs Using Response Plots" on page 8-36
- "Control System Designer Tuning Methods" on page 8-6

Update Simulink Model and Validate Design

This example shows how to update compensator blocks in a Simulink model and validate a control system design.

To tune a control system for a nonlinear Simulink model, **Control System Designer** linearizes the system. Therefore, it is good practice to validate your tuned control system in Simulink.

1. Tune your control system using **Control System Designer**.

   For an example, see "Design Compensator Using Automated PID Tuning and Graphical Bode Design" on page 8-17.
2. Insure that the control system satisfies the design requirements.

   In **Control System Designer**, analyze the controller design. For more information, see "Analyze Designs Using Response Plots" on page 8-36.
3. Write tuned compensator parameters to your Simulink model.

   In **Control System Designer**, on the **Control System** tab, click  **Update Blocks**.
4. Simulate the updated model.

   In the Simulink model window, click  .
5. Verify whether your compensator satisfies the design requirements when simulated with your nonlinear Simulink model.

See Also

Control System Designer

More About

- "Design Compensator Using Automated PID Tuning and Graphical Bode Design" on page 8-17

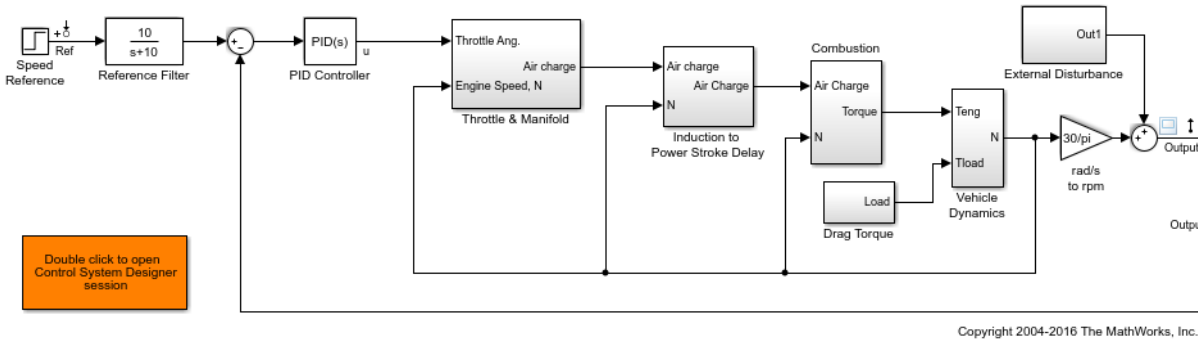
Single Loop Feedback/Prefilter Compensator Design

This example shows how to tune multiple compensators (feedback and prefilter) to control a single loop.

Open the Model

Open the engine speed control model and take a few moments to explore it.

```
open_system('scdspeedctrl');
```



Design Overview

This example introduces the process of designing a single-loop control system with both feedback and prefilter compensators. The goal of the design is to

- Track the reference signal from a Simulink step block `scdspeedctrl/Speed Reference`. The design requirement is to have a settling time of under 5 seconds and zero steady-state error to the step reference input.
- Reject an unmeasured output disturbance specified in the subsystem `scdspeedctrl/External Disturbance`. The design requirement is to reduce the peak deviation to 190 RPM and to have zero steady-state error for a step disturbance input.

In this example, the stabilization of the feedback loop and the rejection of the output disturbance are achieved by designing the PID compensator `scdspeedctrl/PID Controller`. The prefilter `scdspeedctrl/Reference Filter` is used to tune the response of the feedback system to changes in the reference tracking.

Open the Control System Designer

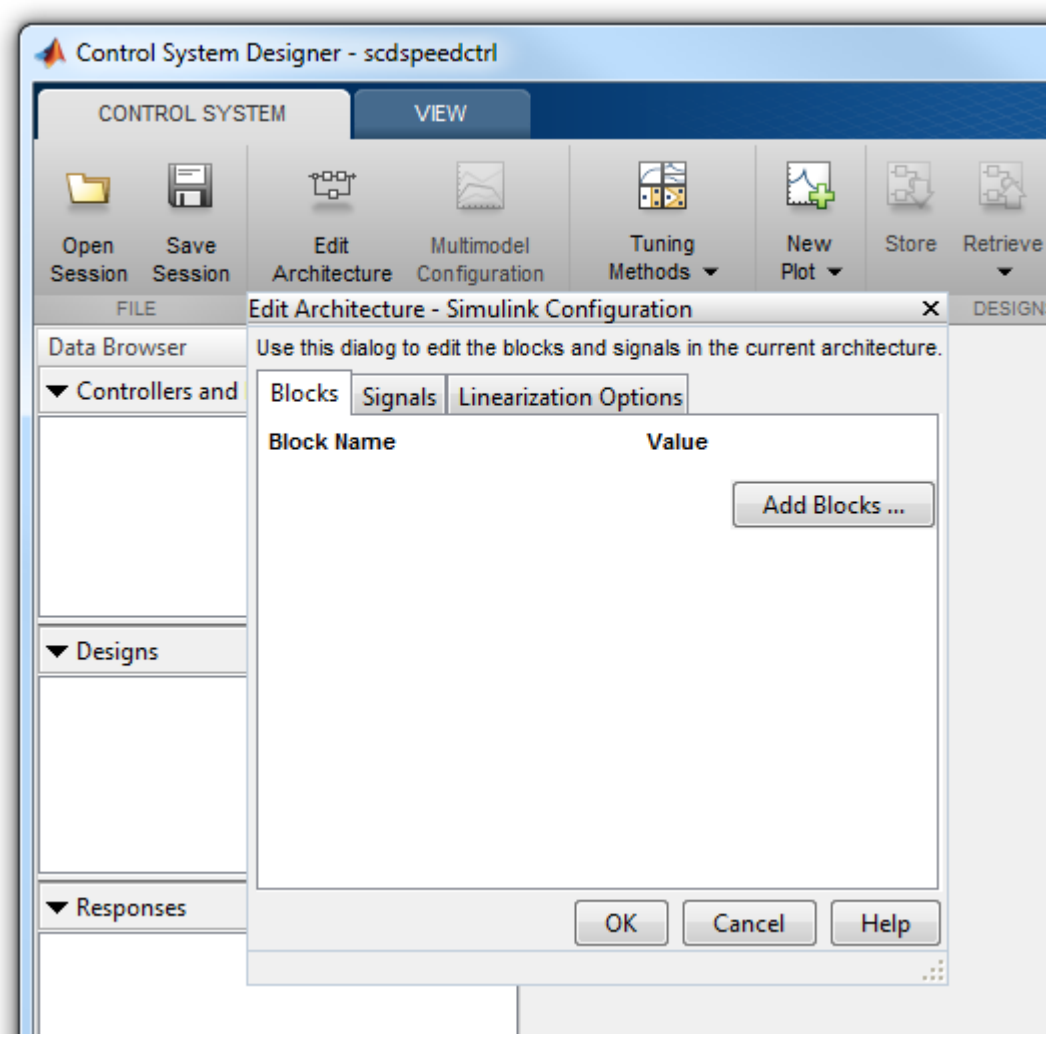
In this example, you will use Control System Designer to tune the compensators in this feedback system. To open the Control System Designer

- Launch a pre-configured Control System Designer session by double-clicking the subsystem in the lower left corner of the model.
- Configure the Control System Designer using the following procedure.

Start a New Design

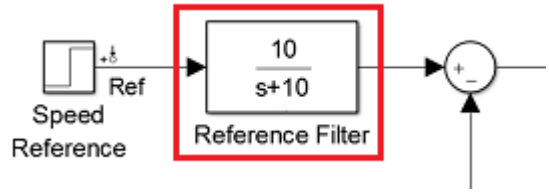
Step 1 To open the Control System Designer, in the Simulink model window, select **Analysis > Control Design > Control System Designer**.

The **Edit Architecture** dialog box opens when the Control System Designer is launched.

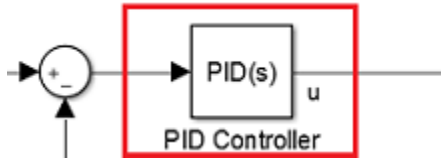


Step 2 In the **Edit Architecture** dialog box, on the **Blocks** tab, click **Add Blocks**, and select the following blocks to tune:

- `scdspeedctrl/Reference Filter`

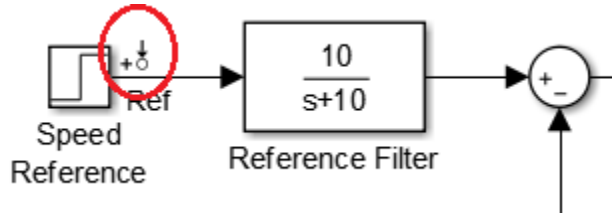


- `scdspeedctrl/PID Controller`



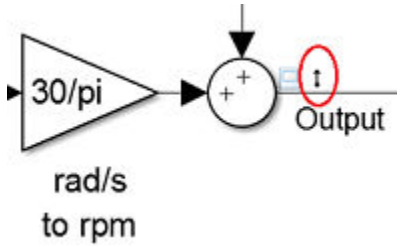
Step 3 On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

- Input: `scdspeedctrl/Speed Reference` output port 1



- Input `scdspeedctrl/External Disturbance/Step Disturbance` output port 1

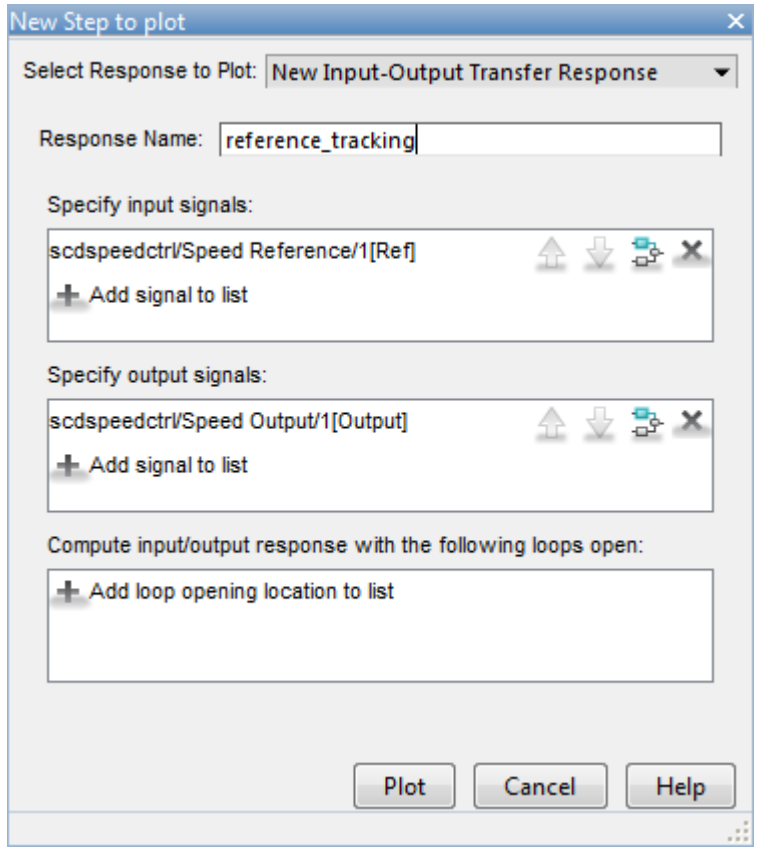
- Output `scdspeedctrl/Speed Output` output port 1



Step 4 On the **Linearization Options** tab, in the **Operating Point** drop-down list, select **Model Initial Condition**.

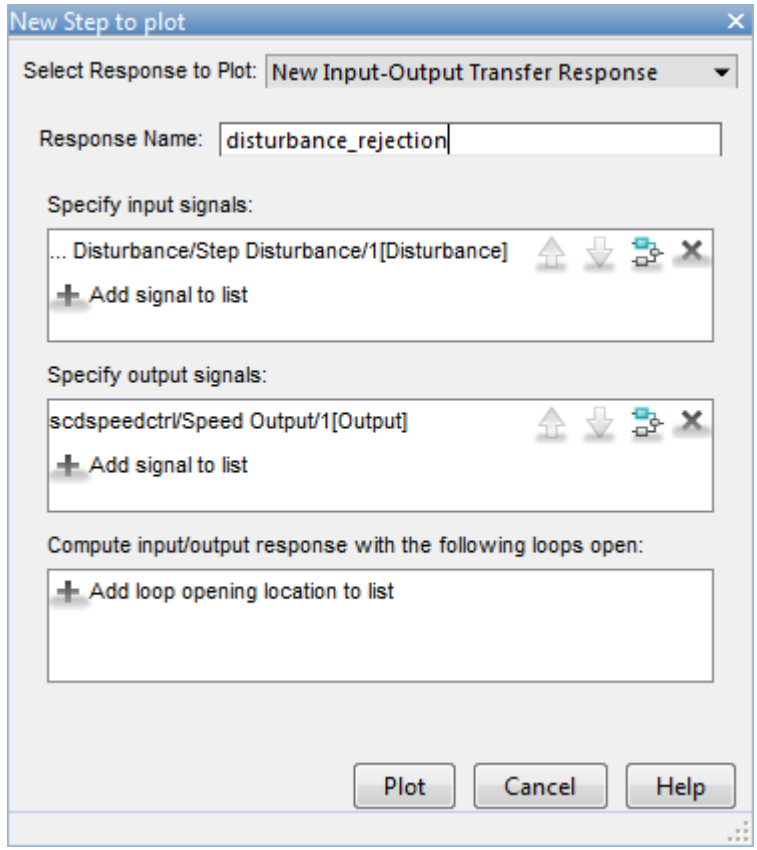
Step 5 Create new plots to view the step responses while tuning the controllers.

- In the Control System Designer, click **New Plot**, and select **New Step**. In the **Select Response to Plot** drop-down menu, select **New Input-Output Transfer Response**. Configure the response as follows:



To view the response, click **Plot**.

- Similarly, create a step response plot to show the disturbance rejection. In the **New Step to plot** dialog box, configure the response as follows::



Tune Compensators

The Control System Designer contains four methods to tune a control system:

- Manually tune the parameters of each compensator using the compensator editor. For more information, see "Tuning Simulink Blocks in the Compensator Editor".
- Graphically tune the compensator poles, zeros, and gains using open/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information,

see "Enforcing Time and Frequency Requirements on a Single-Loop Controller Design".

- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID tuning**, **IMC tuning**, **Loop shaping** (requires Robust Control Toolbox™ software), or **LQG synthesis**.

Completed Design

The following compensator parameters satisfy the design requirements:

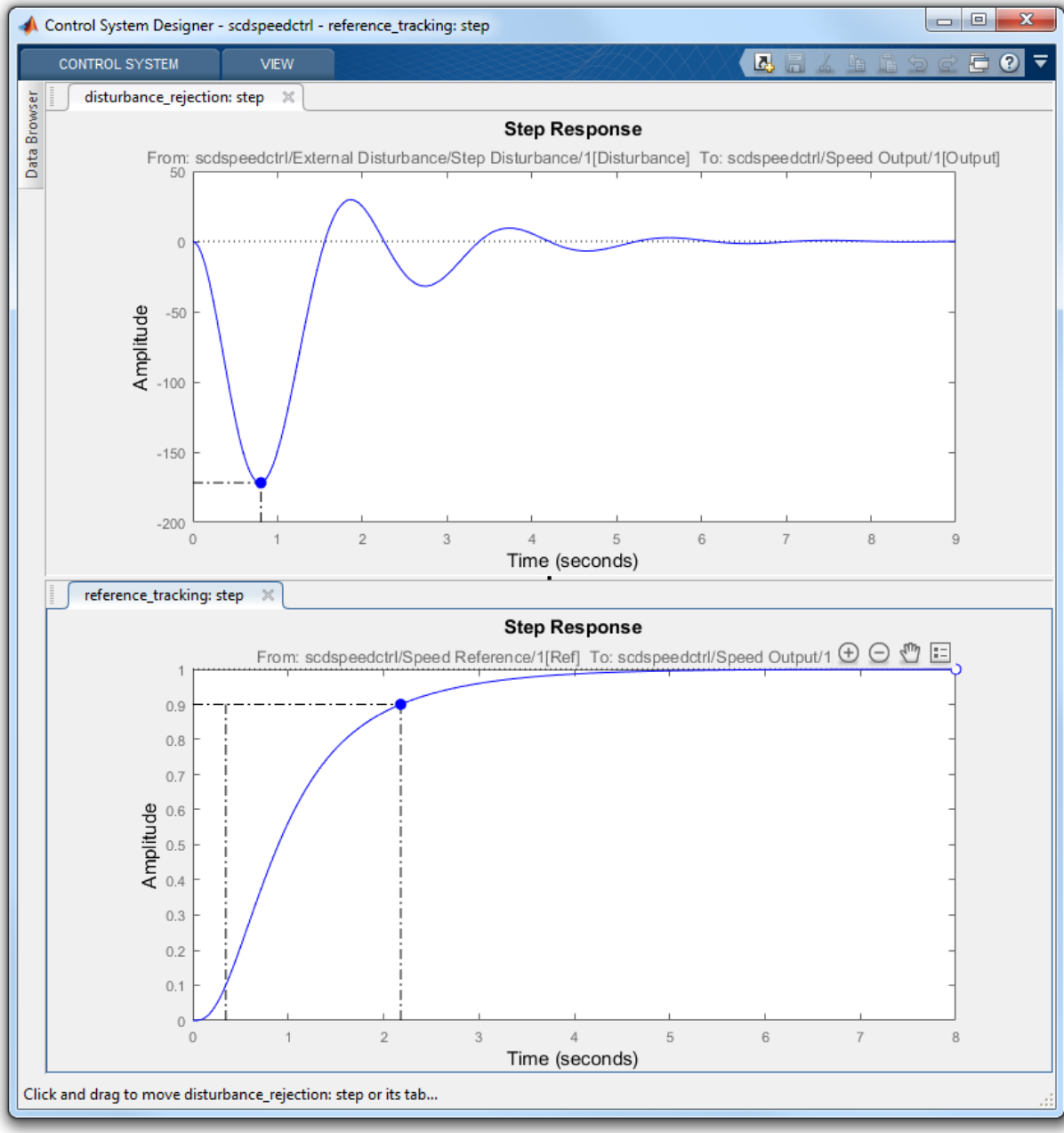
- `scdspeedctrl/PID Controller` has parameters:

```
P = 0.0012191

I = 0.0030038
```

- `scdspeedctrl/Reference Filter`:

```
Numerator = 10;

Denominator = [1 10];
```

The responses of the closed-loop system are shown below:


Control System Designer - scdspeedctrl - reference_tracking: step
CONTROL SYSTEM
VIEW
Data Browser
disturbance_rejection: step
Step Response
From: scdspeedctrl/External Disturbance/Step Disturbance/1[Disturbance] To: scdspeedctrl/Speed Output/1[Output]
Amplitude
Time (seconds)
reference_tracking: step
Step Response
From: scdspeedctrl/Speed Reference/1[Ref] To: scdspeedctrl/Speed Output/1
Amplitude
Time (seconds)
Click and drag to move disturbance_rejection: step or its tab...

Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdspeedctrl')
```

See Also

Control System Designer

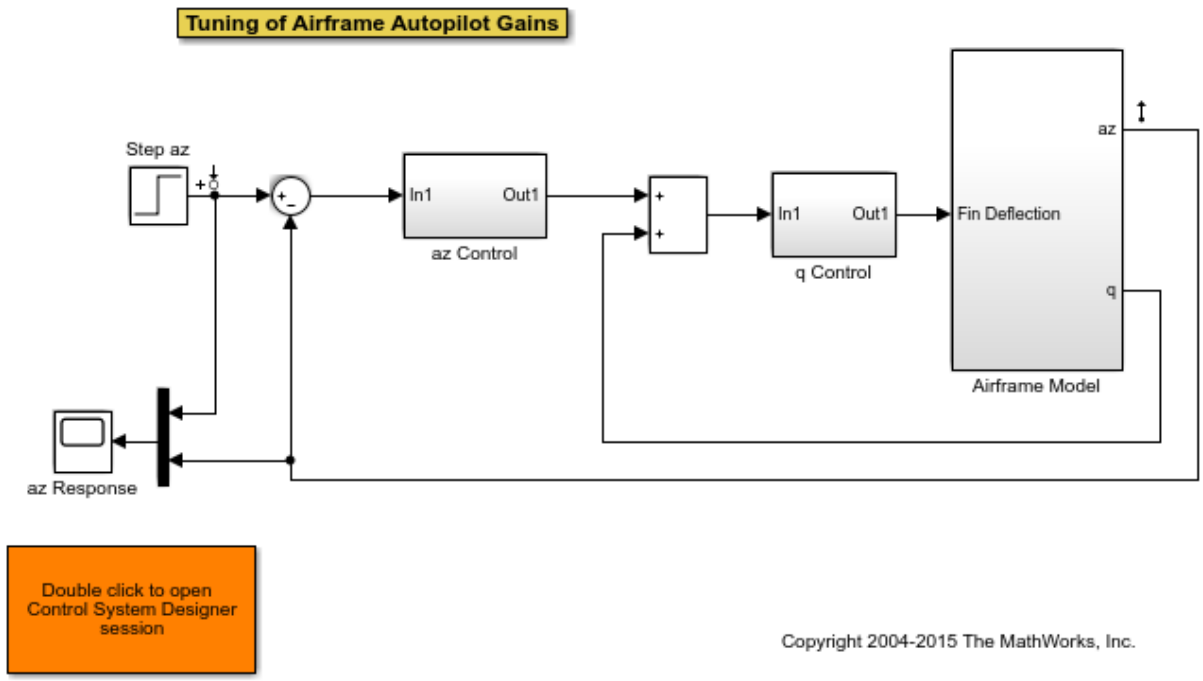
Cascaded Multi-Loop/Multi-Compensator Feedback Design

This example shows how to tune two cascaded feedback loops using Simulink Control Design.

Open the Model

Open the airframe model and take a few moments to explore it.

```
open_system('scdairframectrl');
```



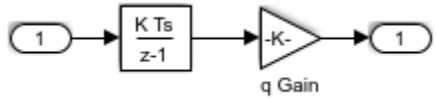
Design Overview

This example introduces the process of designing two cascaded feedback loops so that the acceleration component (az) tracks reference signals with a maximum rise time of 0.5 seconds. The feedback loop structure in this example uses the body rate (q) as an inner feedback loop and the acceleration (az) as an outer feedback loop.

The two feedback controllers are:

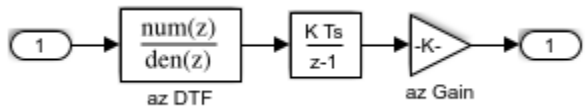
- `scdairframectrl/q Control` - A discrete-time integrator and a gain block stabilize the inner loop.

```
open_system('scdairframectrl/q Control')
```



- `scdairframectrl/az Control` - A discrete-time integrator, a discrete transfer function, and a gain block stabilize the outer loop.

```
open_system('scdairframectrl/az Control')
```



Decoupling Loops in a Multi-Loop Design

The typical design procedure for cascaded feedback systems is to first design the inner loop and then the outer loop. In the Control System Designer it is possible to design both loops simultaneously; by default, when designing a multi-loop feedback system the coupling effects between loops are taken into account. However, when designing two feedback loops simultaneously, it might be necessary to remove the effect of the outer loop when tuning the inner loop. In this example, you design the inner feedback loop (q) with the effect of the outer loop removed (az). The example shows how to decouple feedback loops in the Control System Designer.

Open the Control System Designer

In this example, you will use Control System Designer to tune the compensators in this feedback system. To open the Control System Designer

- Launch a pre-configured Control System Designer session by double-clicking the subsystem in the lower left corner of the model.

- Configure the Control System Designer using the following procedure.

Start a New Design

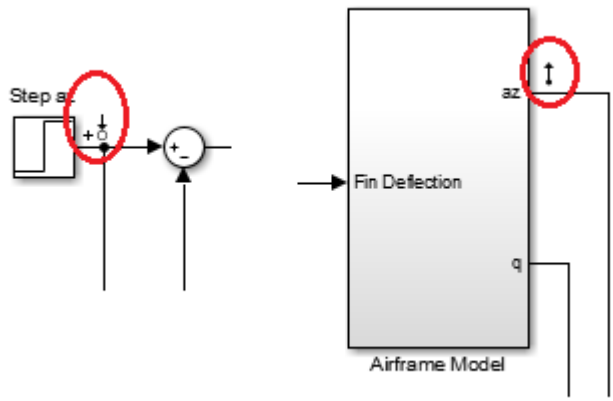
Step 1 To open the Control System Designer, in the Simulink model window, select **Analysis > Control Design > Control System Designer**.

Step 2 In the **Edit Architecture** dialog box, on the **Blocks** tab, select the following blocks to tune:

- `scdairframectrl/q Control/q Gain`
- `scdairframectrl/az Control/az Gain`
- `scdairframectrl/az Control/az DTF`

On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

- Input: `scdairframectrl/Step az` - output port 1
- Output: `scdairframectrl/Airframe Model` - output port 1



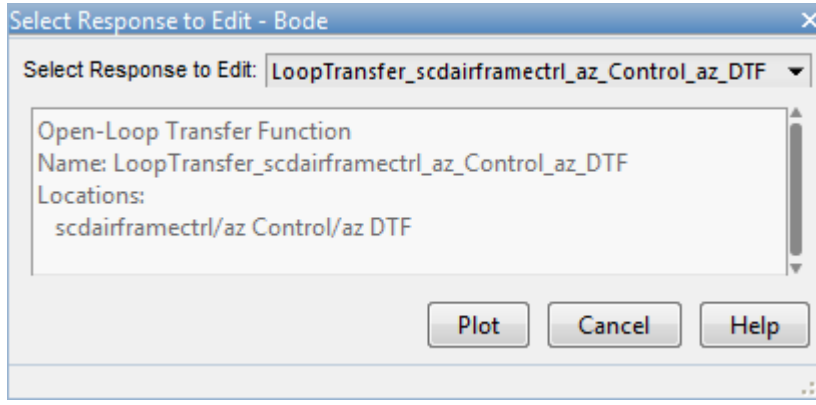
In addition, the following loops are shown in **Data Browser** in the **Responses** area, since they are automatically recognized as potential feedback loops for open-loop design:

- Open Loop at outport 1 of `scdairframectrl/az Control/az DTF`

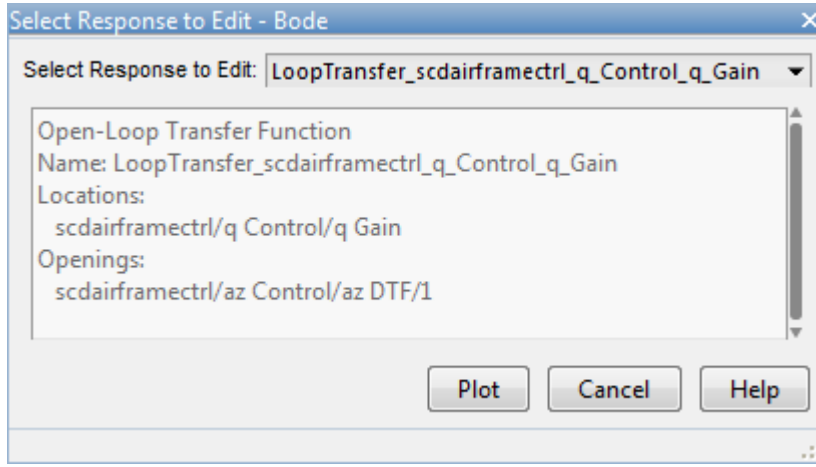
- Open Loop at outport 1 of `scdairframectrl/az Control/az Gain`
- Open Loop at outport 1 of `scdairframectrl/q Control/q Gain`

Step 3 Open graphical Bode editors for each of the following responses. In the Control System Designer, select **Tuning Methods > Bode Editor**. Then, in the **Select Response to Edit** drop-down list, select the corresponding open-loop responses.

- Open Loop at outport 1 of `scdairframectrl/az Control/az DTF`

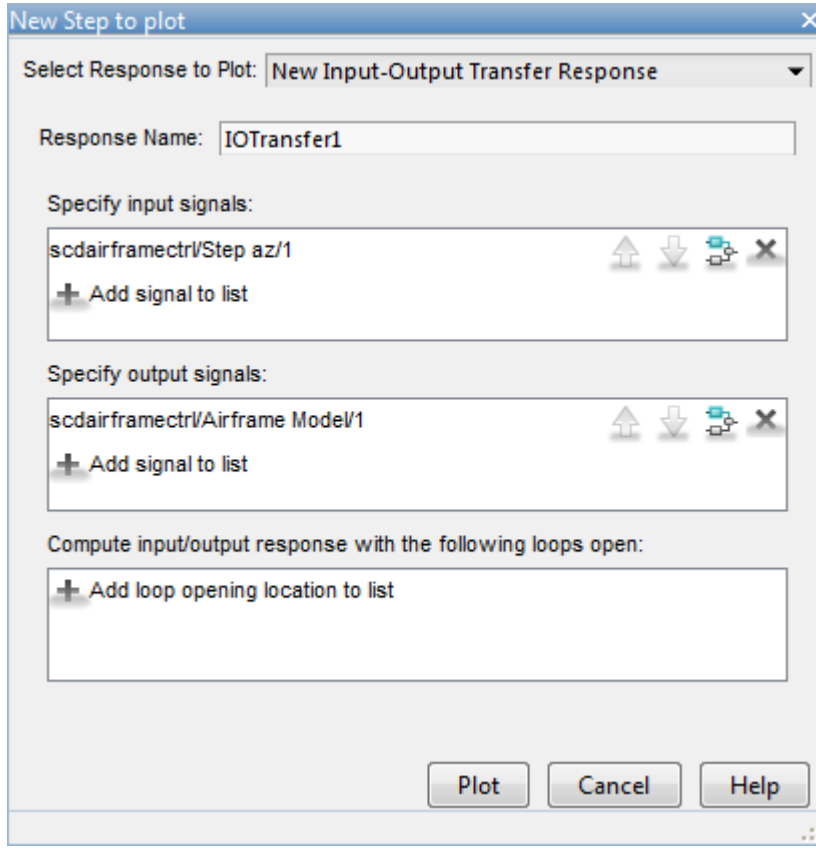


- Open Loop at outport 1 of `scdairframectrl/q Control/q Gain`



Step 4 To view the closed-loop response of the feedback system, create a step plot for a new input-output transfer function response. Select **New Plot > New Step**, and in the **Select Response to Plot** drop-down list, select **New Input-Output Transfer Response**.

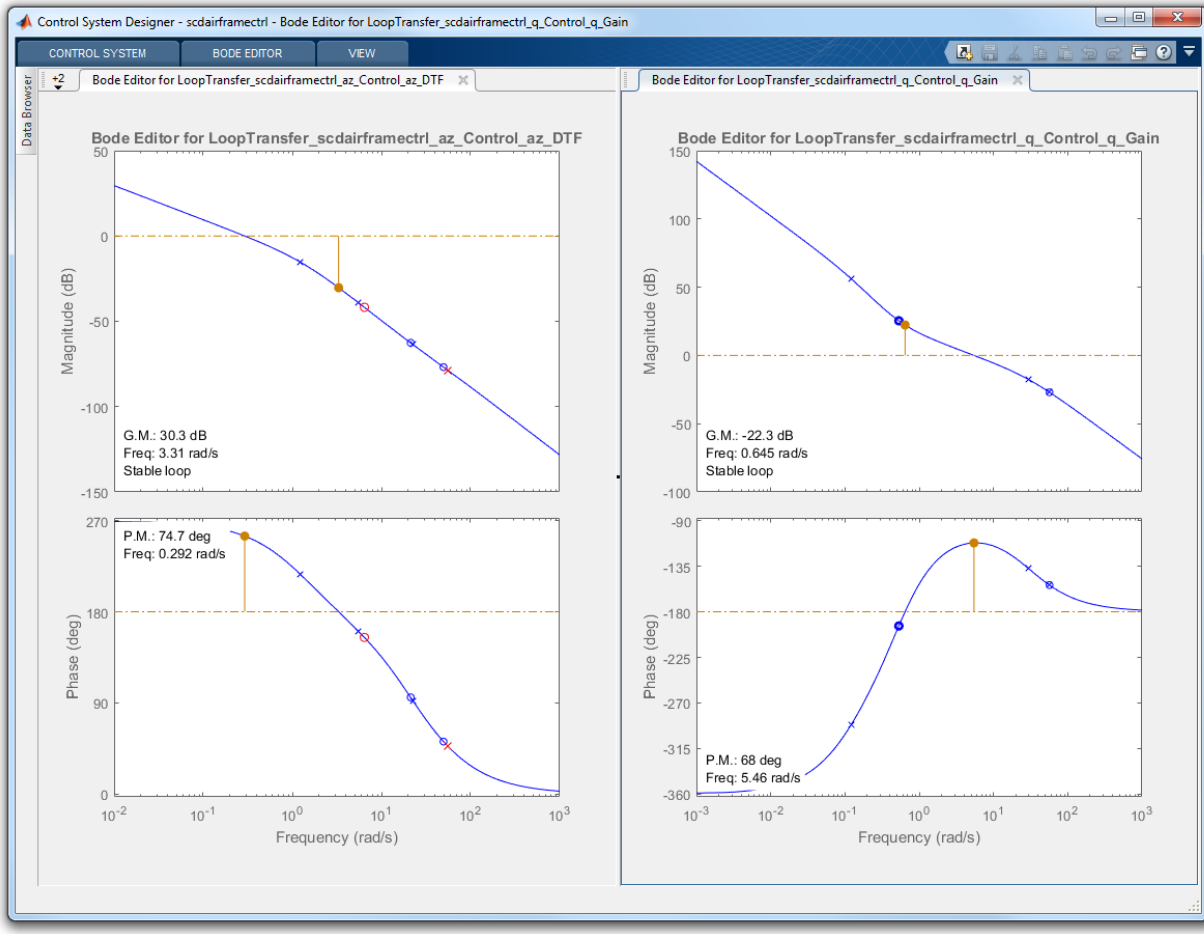
- Add `scdairframectrl/Step az/1` as an input signal and `scdairframectrl/Airframe Model/1` as an output signal.



Removing Effect of Outer Feedback Loop

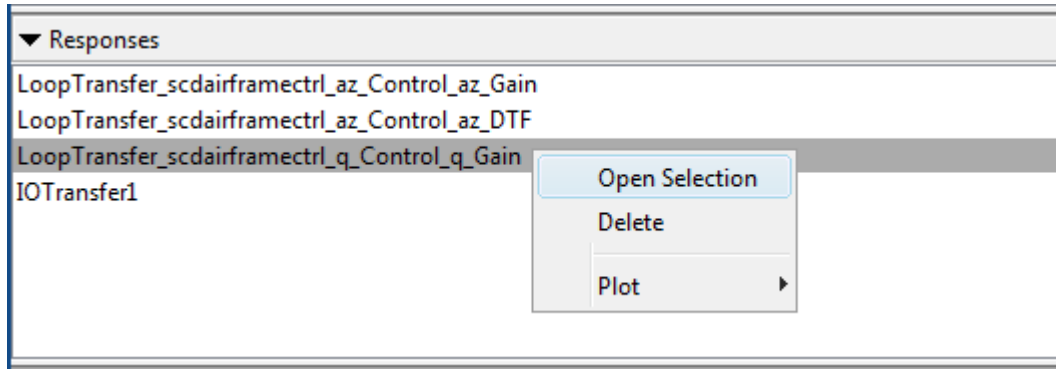
In the outer-loop bode editor plot, **Bode Editor for LoopTransfer_scdairframectrl_az_Control_az_DTF**, increase the gain of the feedback loop, by dragging the magnitude response upward. The inner-loop bode editor

plot, **Bode Editor for LoopTransfer_scdairframectrl_q_Control_q_Gain**, plot also changes. This is a result of the coupling between the feedback loops. A more systematic approach is to first design the inner feedback loop, with the outer loop open.

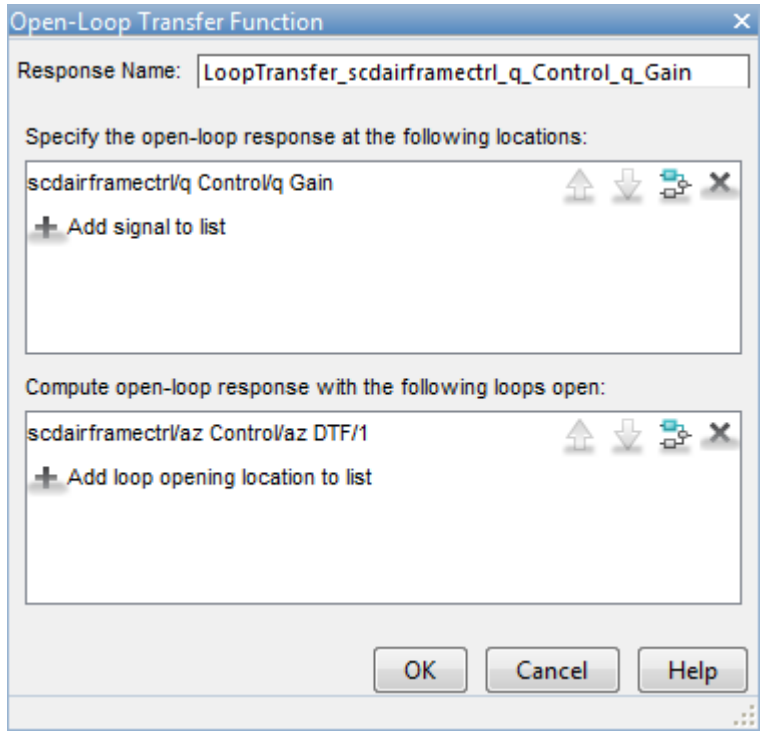


To remove the effect of the outer loop when designing the inner loop, add a loop opening to the open-loop response of the inner loop.

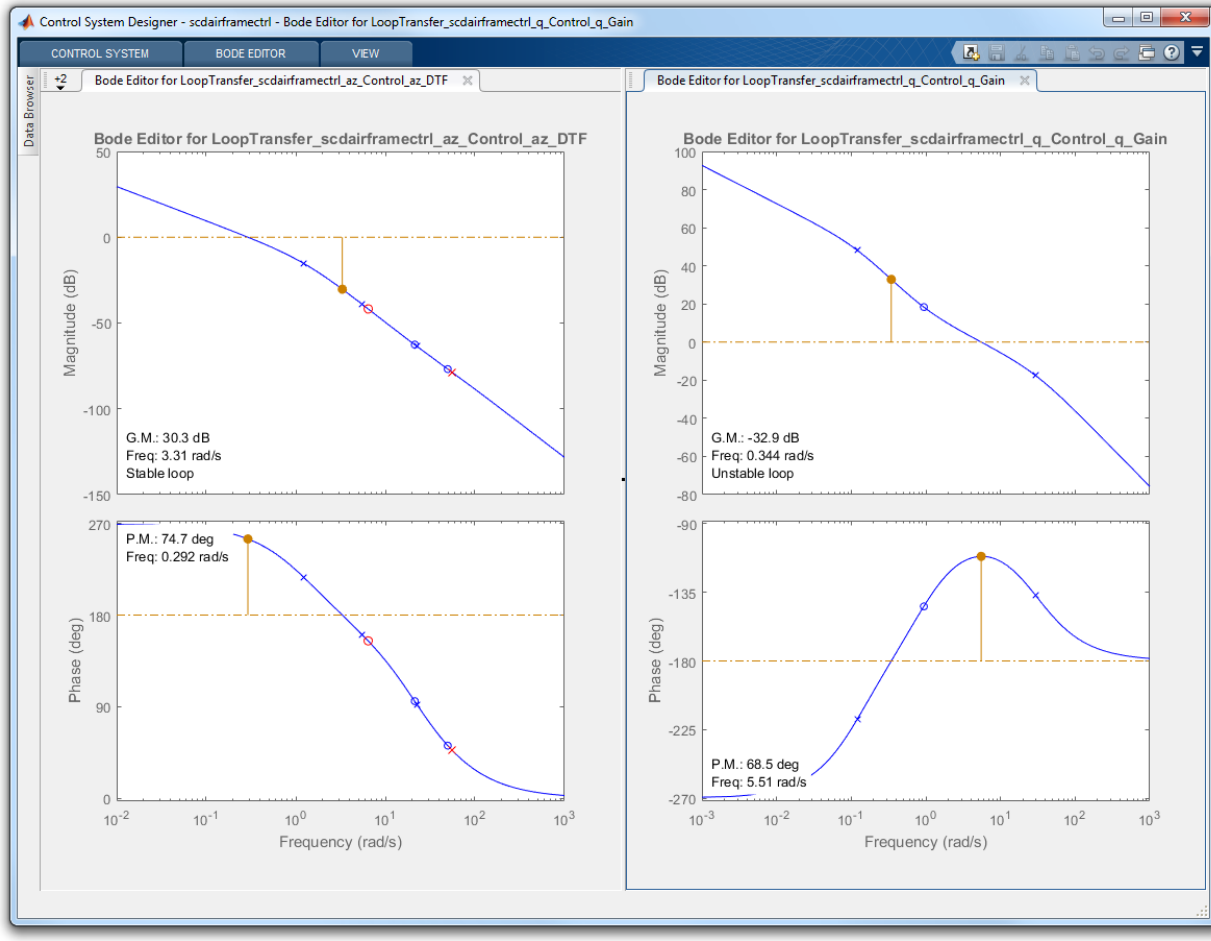
Step 1 In the **Data Browser**, in the **Responses** area, right-click on the inner loop response, and select **Open Selection**.



Step 2 In the Open-Loop Transfer Function dialog box, specify `scdairframectrl/az Control/az DTF/1` as the loop opening. Click **OK**.



Step 3 In the outer-loop bode editor plot, increase the gain by dragging the magnitude response. Since the loops are decoupled, the inner-loop bode editor plot does not change.



You can now complete the design of the inner loop without the effect of the outer loop and simultaneously design the outer loop while taking the effect of the inner loop into account.

Tune Compensators

The Control System Designer contains four methods to tune a control system:

- Manually tune the parameters of each compensator using the compensator editor. For more information, see "Tuning Simulink Blocks in the Compensator Editor".
- Graphically tune the compensator poles, zeros, and gains using open/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information, see "Enforcing Time and Frequency Requirements on a Single-Loop Controller Design".
- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID tuning**, **IMC tuning**, **Loop shaping** (requires Robust Control Toolbox™ software), or **LQG synthesis**.

Complete Design

The following compensator parameters satisfy the design requirements:

- `scdairframectrl/q Control/q Gain`:

```
K_q = 2.7717622
```

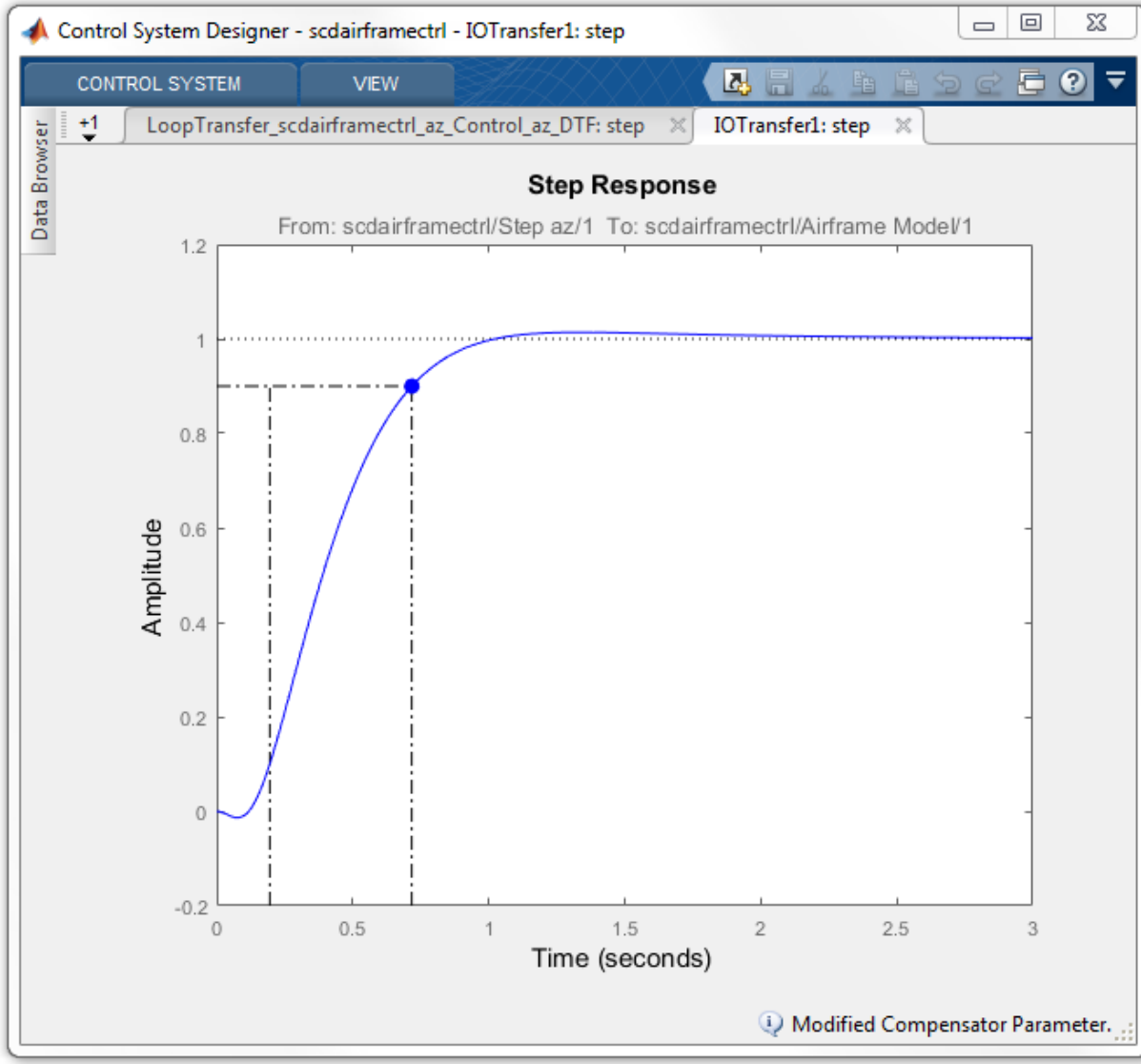
- `scdairframectrl/az Control/az Gain`

```
K_az = 0.00027507
```

- `scdairframectrl/az Control/az DTF`

```
Numerator = [100.109745 -99.109745]

Denominator = [1 -0.88893]
```

The response of the closed-loop system is shown below:


Control System Designer - scdairframectrl - IOTransfer1: step
CONTROL SYSTEM
VIEW
Data Browser
+1
LoopTransfer_scdairframectrl_az_Control_az_DTF: step
IOTransfer1: step
Step Response
From: scdairframectrl/Step az/1 To: scdairframectrl/Airframe Model/1
Amplitude
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.5
1
1.5
2
2.5
3
Time (seconds)
Modified Compensator Parameter.

Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdairframectrl')
```

See Also

Control System Designer

More About

- “Design Multiloop Control System” (Control System Toolbox)

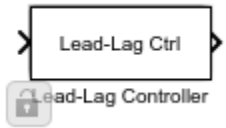
Tune Custom Masked Subsystems

This example shows how to enable custom masked subsystems in Control System Designer. Once configured, you can tune a custom masked subsystem in the same way as any supported blocks in Simulink Control Design. For more information, see "What Blocks Are Tunable?".

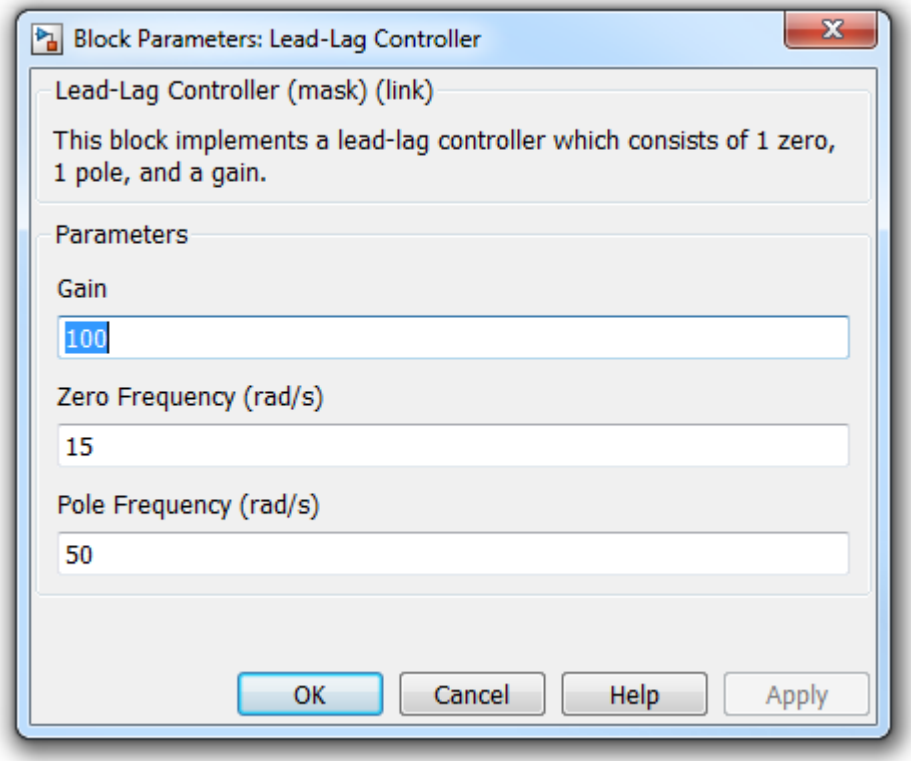
Lead-Lag Library Block

For this example, tune the `Lead-Lag Controller block` in the `scdexblks` library.

```
open_system('scdexblks')
```



This block implements a compensator with a single zero, a single pole, and a gain. To open the Block Parameters dialog box, add the Lead-Lag Controller block to your model, and double-click the block.



The block uses the specified Gain, `K`, Zero Frequency, `wz`, and Pole Frequency, `wp`, to implement the compensator transfer function:

$$G(s) = K\frac{\frac{s}{wz}+1}{\frac{s}{wp}+1}$$

Configure the Subsystem for Control System Designer

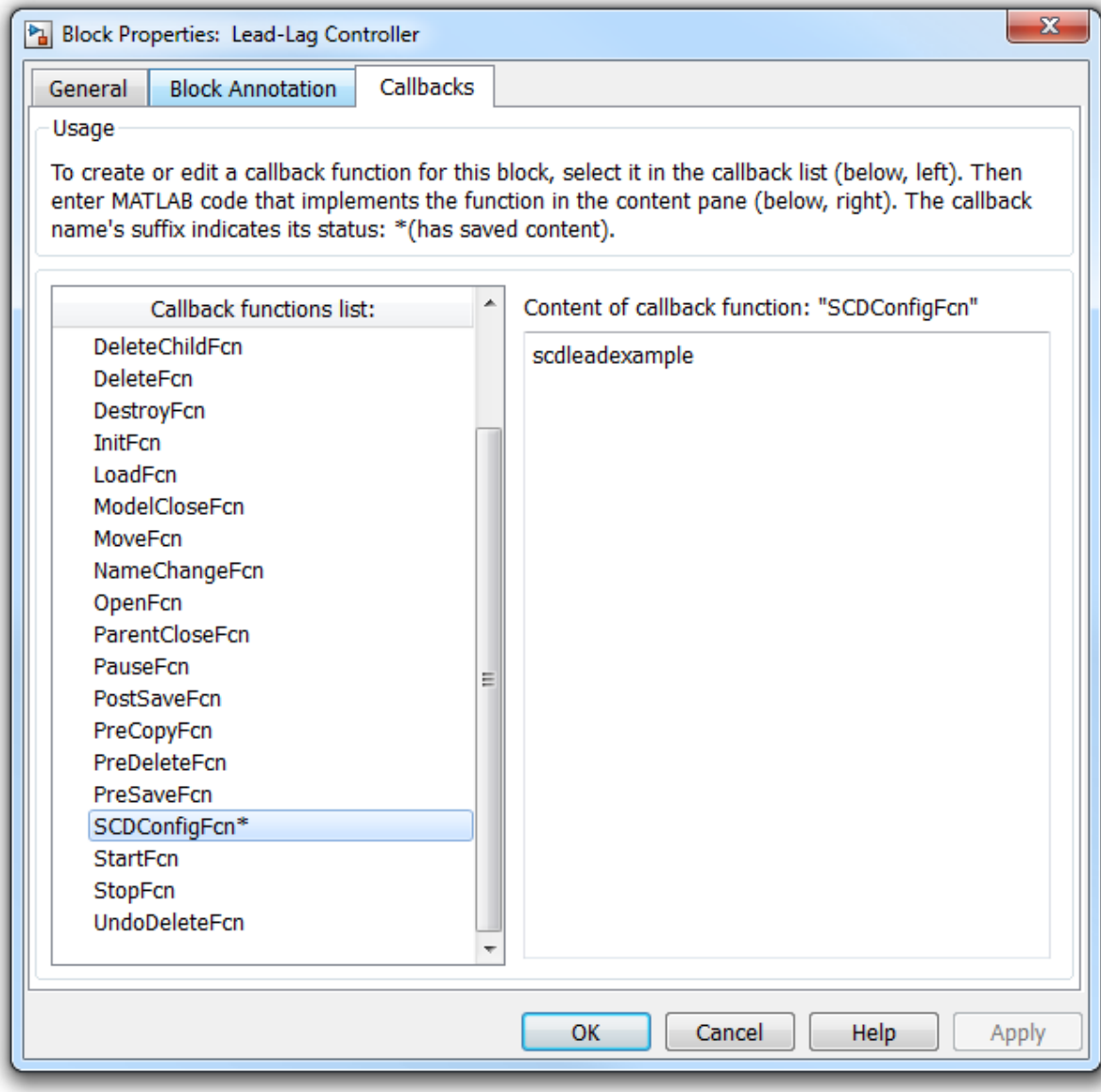
To configure a masked subsystem for tuning with Control System Designer, you specify a configuration function. In this example, use the configuration function in scdleadexample.m, which specifies that:

- There is only one pole allowed (MaxPoles constraint)

- There is only one zero allowed (MaxZeros constraint)
- The gain is tunable (isStaticGainTunable constraint)

Register the configuration in the subsystem using the `SCDConfigFcn` block callback function. Right-click the Lead-Lag Controller block and select **Properties**. In the Block Properties dialog box, on the **Callbacks** tab, set `SCDConfigFcn`.

Alternatively, you can set `SCDConfigFcn` using the command `set_param`.

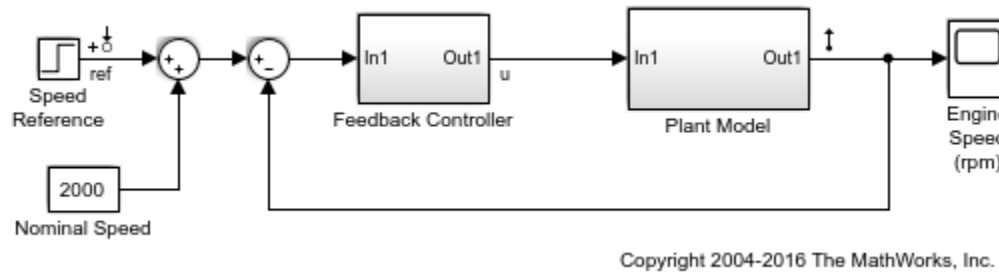

Block Properties: Lead-Lag Controller
General
Block Annotation
Callbacks
Usage
To create or edit a callback function for this block, select it in the callback list (below, left). Then enter MATLAB code that implements the function in the content pane (below, right). The callback name's suffix indicates its status: *(has saved content).
Callback functions list:
DeleteChildFcn
DeleteFcn
DestroyFcn
InitFcn
LoadFcn
ModelCloseFcn
MoveFcn
NameChangeFcn
OpenFcn
ParentCloseFcn
PauseFcn
PostSaveFcn
PreCopyFcn
PreDeleteFcn
PreSaveFcn
SCDConfigFcn*
StartFcn
StopFcn
UndoDeleteFcn
Content of callback function: "SCDConfigFcn"
scdleadexample
OK
Cancel
Help
Apply

After setting the SCDConfigFcn the block is now ready to be used in a Simulink Compensator Design Task.

Example

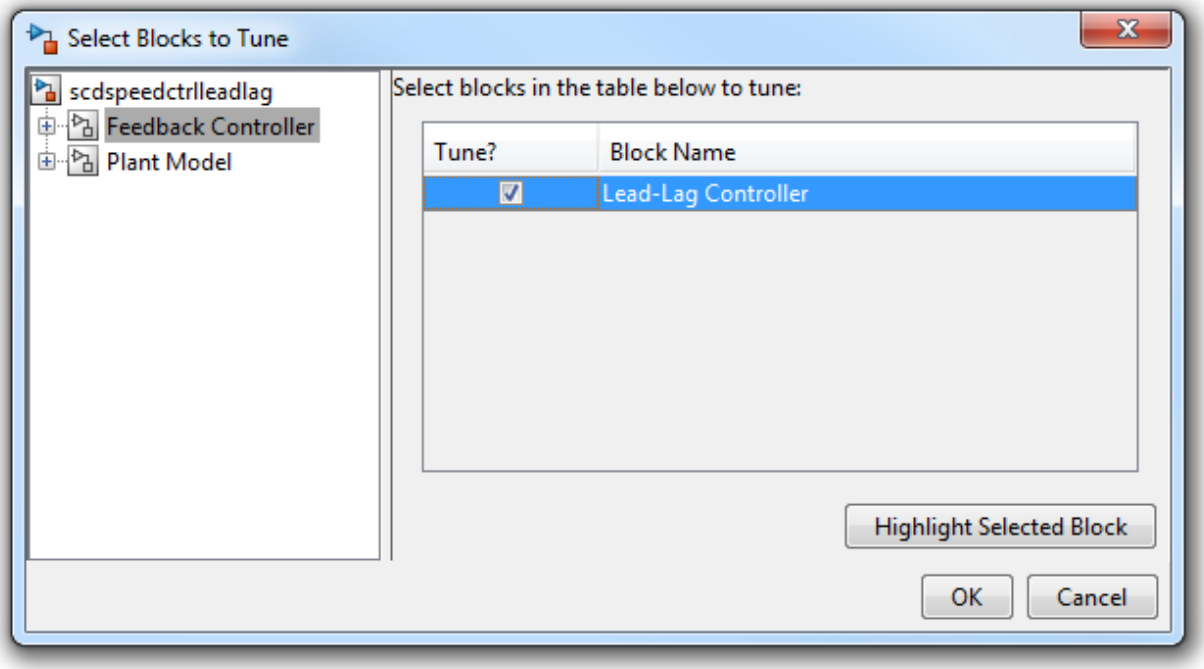
The scdspeedctrlleadlag model uses the Lead-Lag Controller block to tune the feedback loop in "Single Loop Feedback/Prefilter Design". In this model, the SCDConfigFcn property is already set. .

```
open_system('scdspeedctrlleadlag')
```



Step 1 To open the Control System Designer, in the Simulink model window, select **Analysis > Control Design > Control System Designer**.

Step 2 In the **Edit Architecture** dialog box, on the Blocks tab, click **Add Blocks**. In the **Select Blocks to Tune** dialog box, click Feedback Controller, and select `Lead-Lag Controller`.



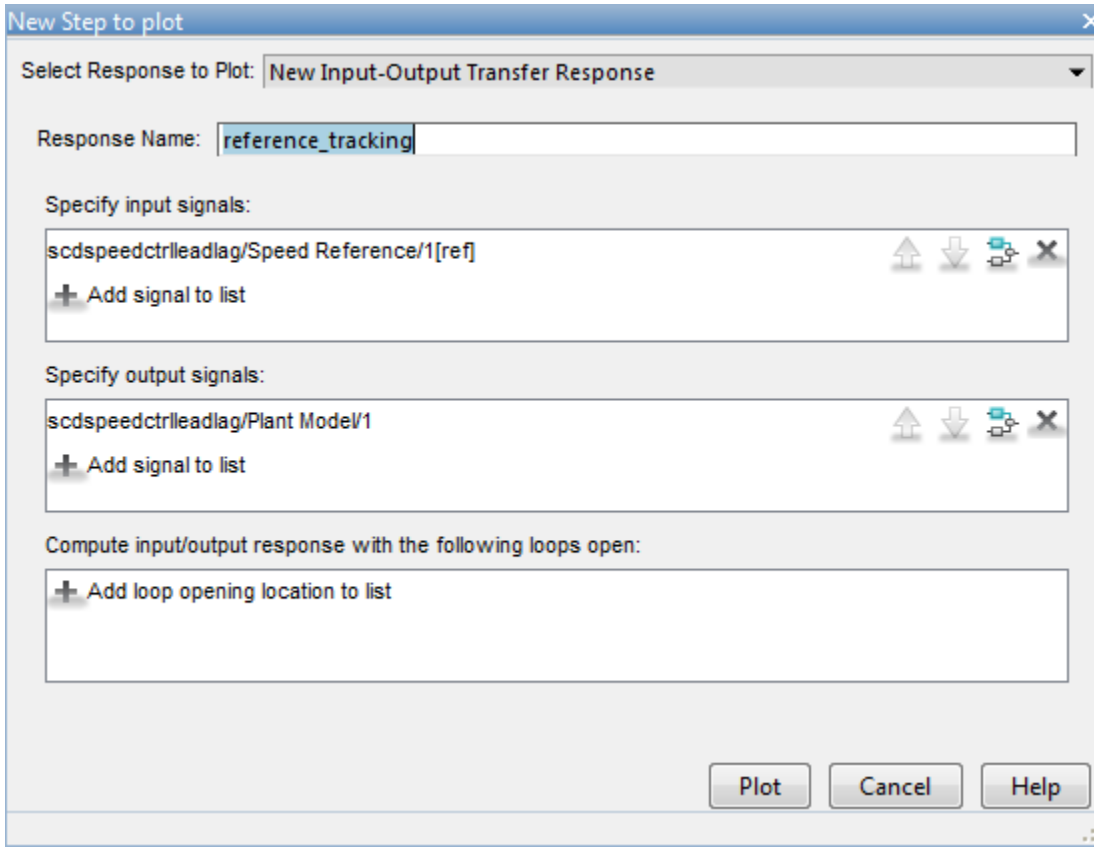
Step 3 On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

- Input: `scdspeedctrlleadlag/Speed Reference` output port 1
- Output `scdspeedctrlleadlag/Plant Model` output port 1

Step 4 On the **Linearization Options** tab, in the **Operating Point** drop-down list, select **Model Initial Condition**.

Step 5 Create new plots to view the step responses while tuning the controllers.

- In the Control System Designer, click **New Plot**, and select **New Step**. In the **Select Response to Plot** drop-down menu, select **New Input-Output Transfer Response**. Configure the response as follows:

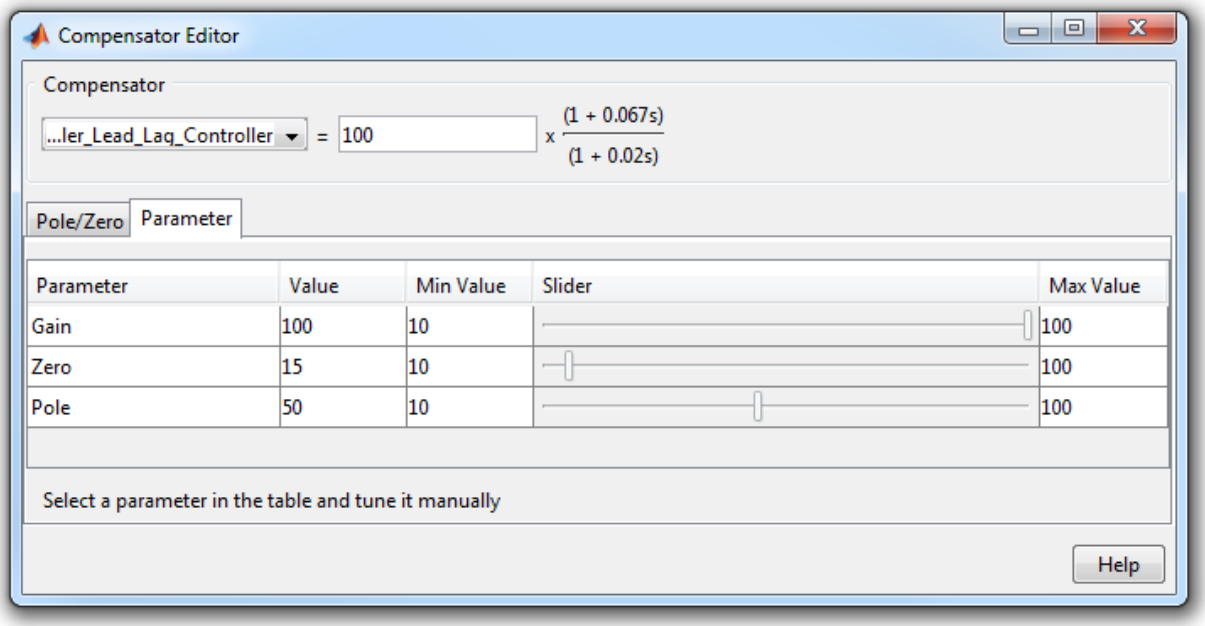


To view the response, click **Plot**.

Tune Compensators

The Control System Designer contains four methods to tune a control system:

- Manually tune the parameters of the `Lead-Lag Controller` using the compensator editor. For more information, see "Tuning Simulink Blocks in the Compensator Editor".



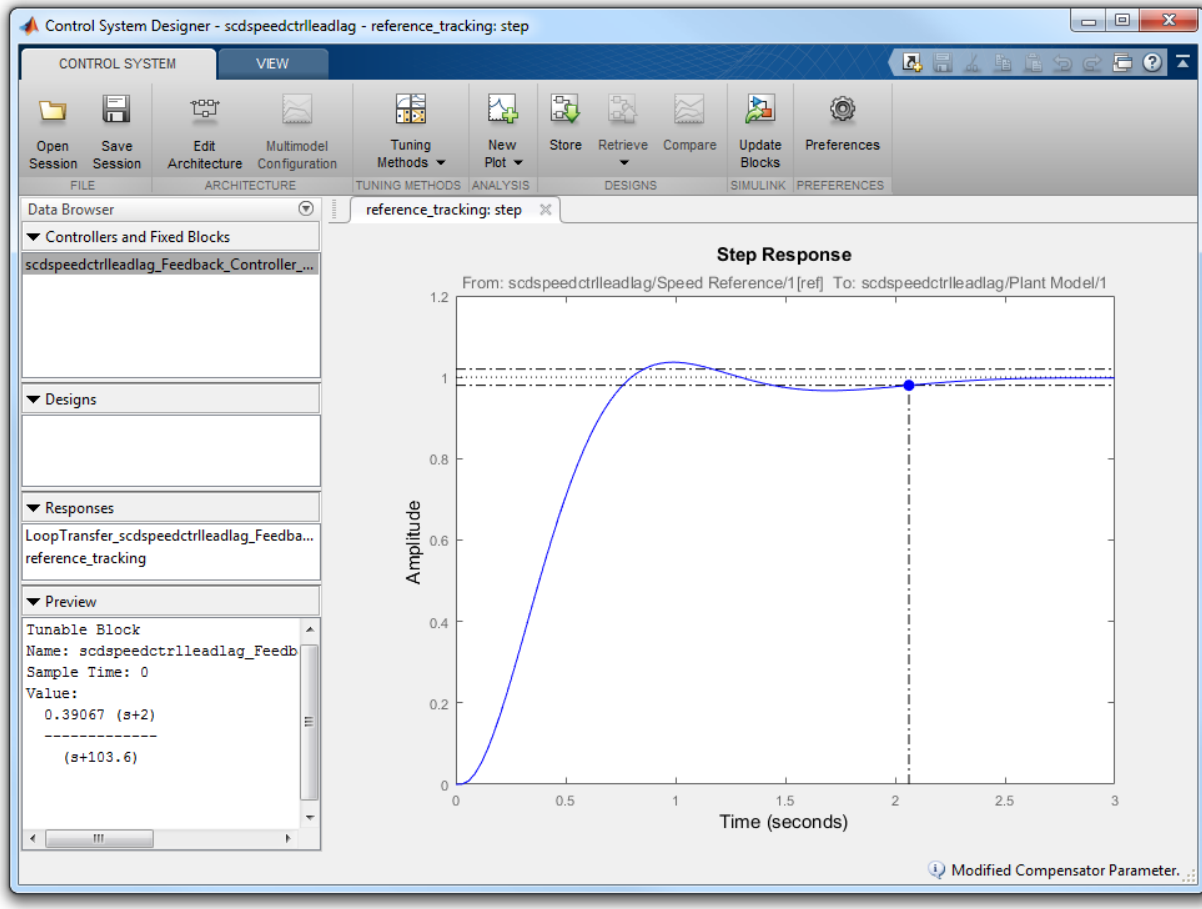
- Graphically tune the compensator poles, zeros, and gains using open/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information, see "Enforcing Time and Frequency Requirements on a Single-Loop Controller Design".
- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID tuning**, **IMC tuning**, **Loop shaping** (requires Robust Control Toolbox™ software), or **LQG synthesis**.

Complete Design

The design requirements for the reference step response in the example "Single Loop Feedback/Prefilter Design" can be met with the following Lead-Lag Controller block parameters:

```
Gain = 0.0075426

Zero Frequency (rad/s) = 2

Pole Frequency (rad/s) = 103.59
```

The responses of the closed-loop system are shown below:



Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdexblks')
bdclose('scdspeedctrlleadlag')
```

See Also

Control System Designer

Tuning Simulink Blocks in the Compensator Editor

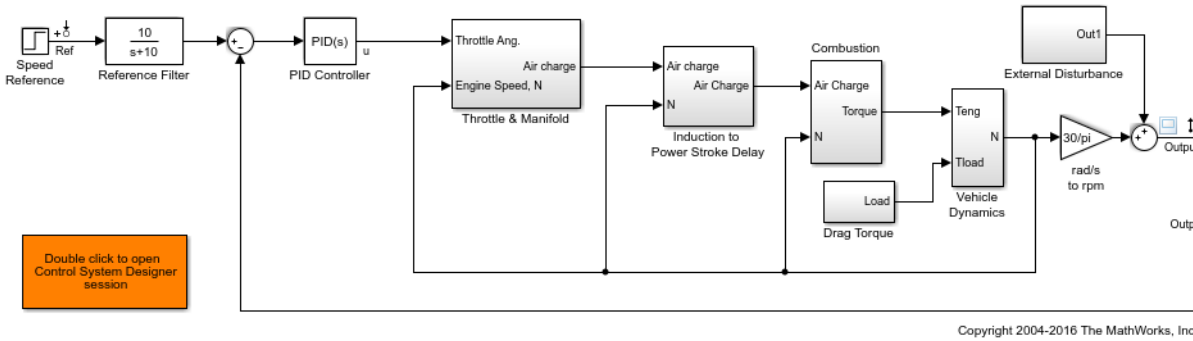
This example shows how to use Compensator Editor dialog box to tune Simulink blocks.

Open the Model

This example uses a model of speed control system for a sparking ignition engine. The initial compensator has been designed in a fashion similar to the example entitled "Single Loop Feedback/Prefilter Design". Take a few moments to explore the model.

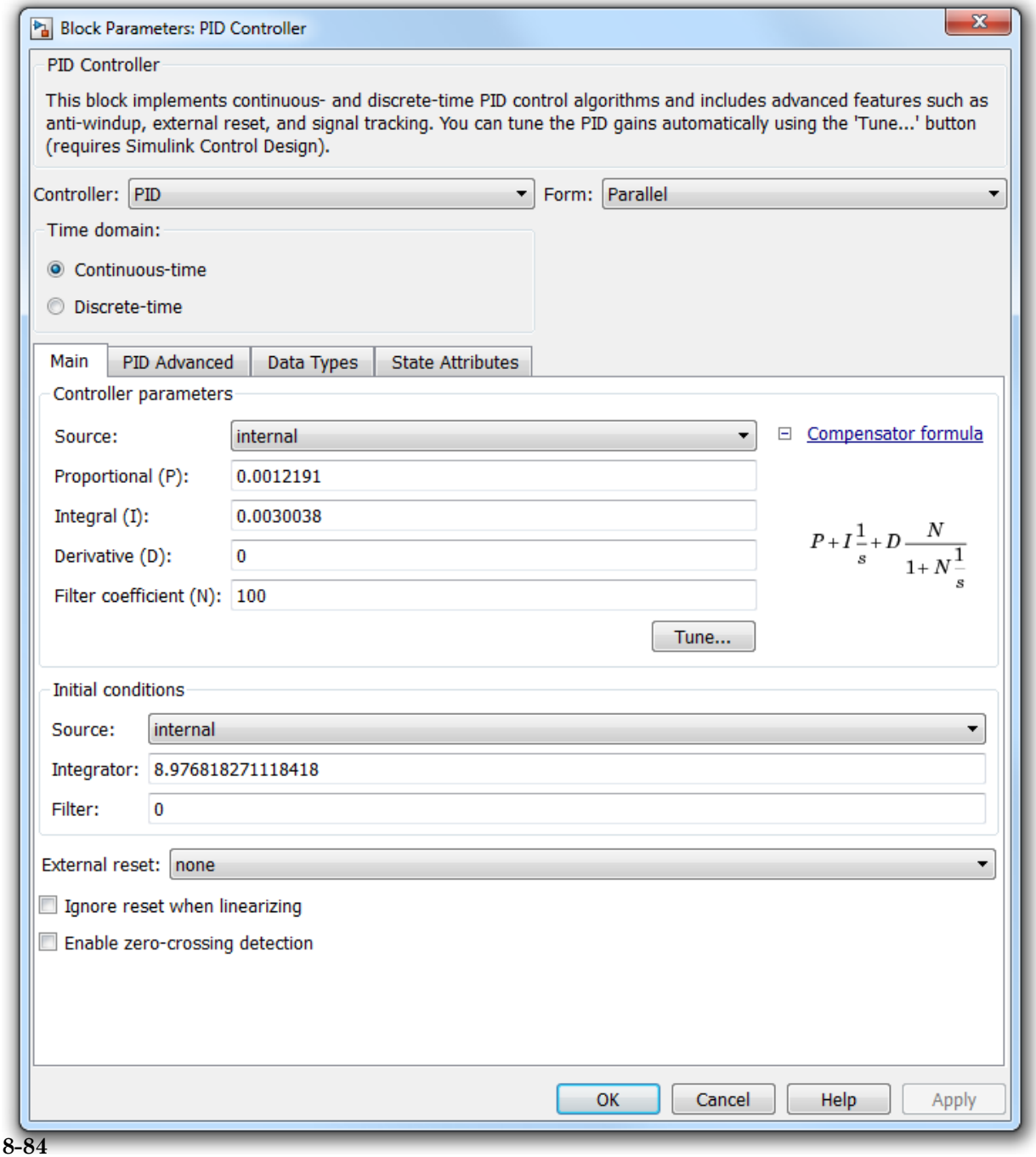
Open the engine speed control model.

```
open_system('scdspeedctrl');
```



Introduction

This example uses the **Compensator Editor** to tune Simulink blocks. When tuning a block in Simulink using the Control System Designer, you can use one of compensator parameter representations. These representations are the block parameters and the pole/zero/gain representations. For example, in the speed control example there is a PID controller with filtered derivative `scdspeedctrl/PID Controller`:


Block Parameters: PID Controller
PID Controller
This block implements continuous- and discrete-time PID control algorithms and includes advanced features such as anti-windup, external reset, and signal tracking. You can tune the PID gains automatically using the 'Tune...' button (requires Simulink Control Design).
Controller: PID
Form: Parallel
Time domain:
Continuous-time
Discrete-time
Main
PID Advanced
Data Types
State Attributes
Controller parameters
Source: internal
Compensator formula
Proportional (P): 0.0012191
Integral (I): 0.0030038
Derivative (D): 0
Filter coefficient (N): 100
$P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}}$
Tune...
Initial conditions
Source: internal
Integrator: 8.976818271118418
Filter: 0
External reset: none
Ignore reset when linearizing
Enable zero-crossing detection
OK
Cancel
Help
Apply

This block implements the traditional PID with filtered derivative as:

$$G(s) = P + \frac{I}{s} + \frac{Ds}{Ns+1}$$

In this block `P`, `I`, `D`, and `N` are the parameters that are available for tuning. Another approach is to reformulate the block transfer function to use zero-pole-gain format:

$$G(s) = \frac{Ps(Ns+1) + I(Ns+1) + Ds^s}{s(Ns+1)} = \frac{K(s^2 + 2\zeta\omega_n + w_n^2)}{s(s+z)}$$

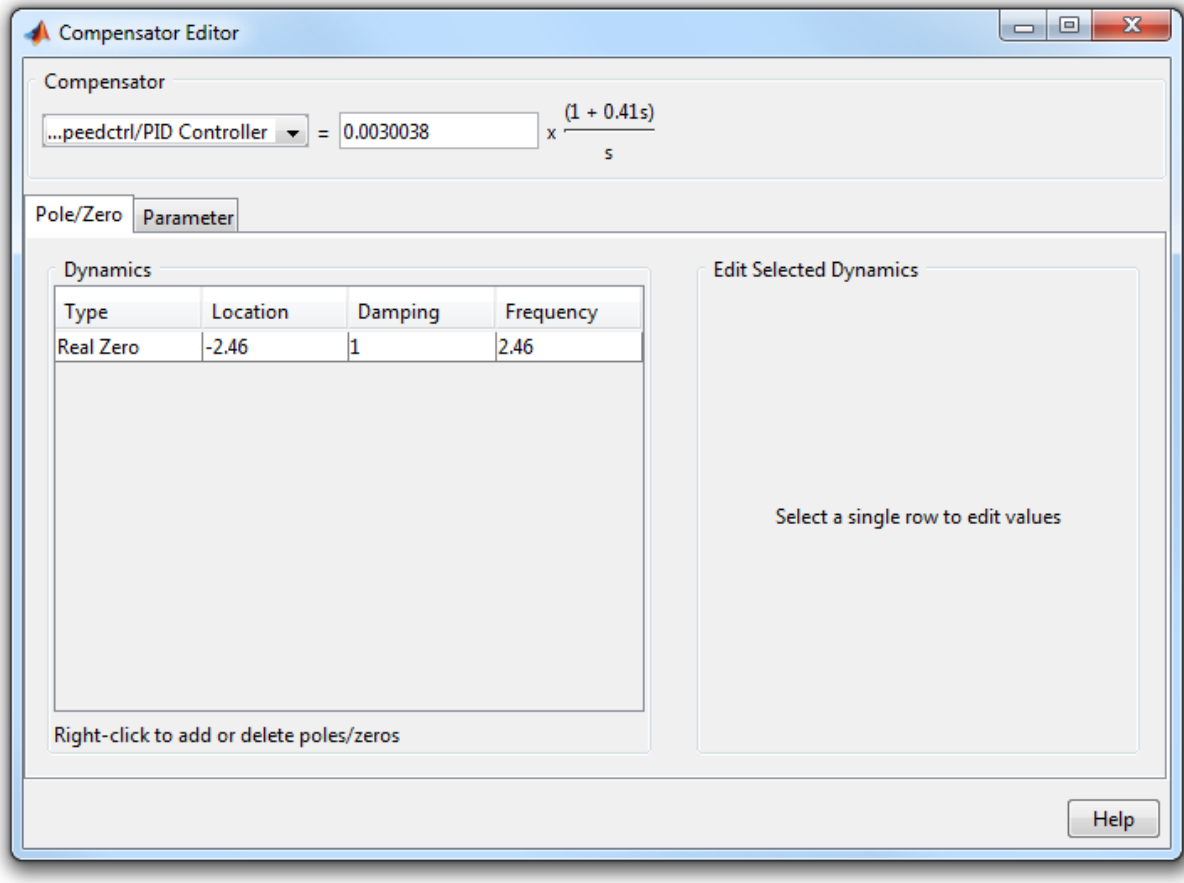
This formulation of poles, zeros, and gains allows for direct graphical tuning on design plots such as Bode, root locus, and Nichols plots. Additionally, the Control System Designer allows for both representations to be tuned using the compensator editor. The tuning of both representations is available for all supported blocks in Simulink Control Design. For more information, see "What Blocks Are Tunable?"

Open Control System Designer

In this example, to tune the compensators in this feedback system, open a pre-configured Control System Designer session by double-clicking the subsystem in the lower left hand corner of the model.

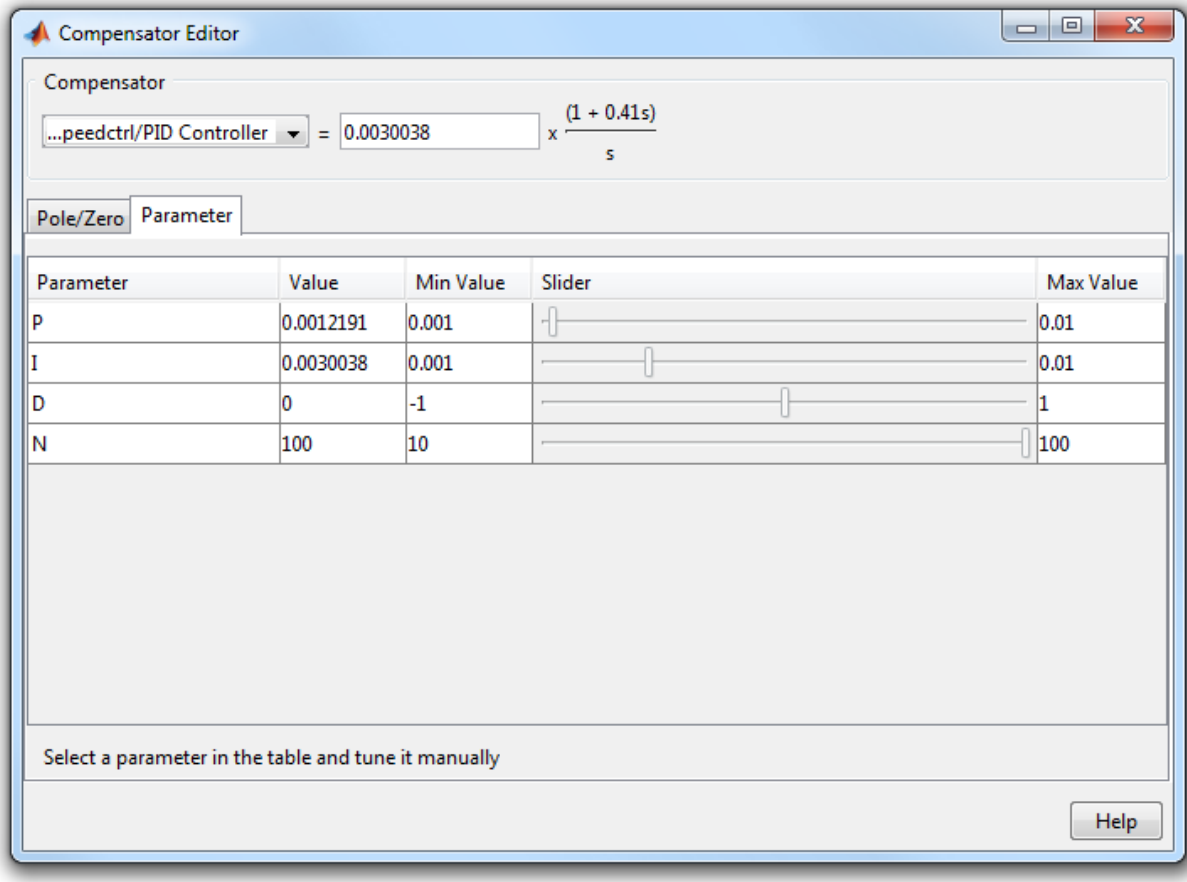
Compensator Editor Dialog Box

You can view the representations of the PID compensator using the Compensator Editor dialog box. To open the **Compensator Editor**, in the **Data Browser**, in the **Controllers and Fixed Blocks area**, double-click `scdspeedctrl_PID_Controller`. In the Compensator Editor dialog box, in the Compensator section, you can view and edit any of the compensators in your system.



On the **Pole/Zero** tab, you can add, delete, and edit compensator poles and zeros. Since the PID with filtered derivative is fixed in structure, the number of poles and zeros is limited to having up to two zeros, one pole, and an integrator at `s = 0`.

On the Parameter tab, you can independently tune the `P`, `I`, `D`, and `N` parameters.



Enter new parameters values in the **Value** column. To interactively tune the parameters, use the sliders. You can change the slider limits using the **Min Value** and **Max Value** columns.

When you change parameter values, any associated editor and analysis plots automatically update.

Complete Design

The design requirements in the example "Single Loop Feedback/Prefilter Design" can be met with the following controller parameters:

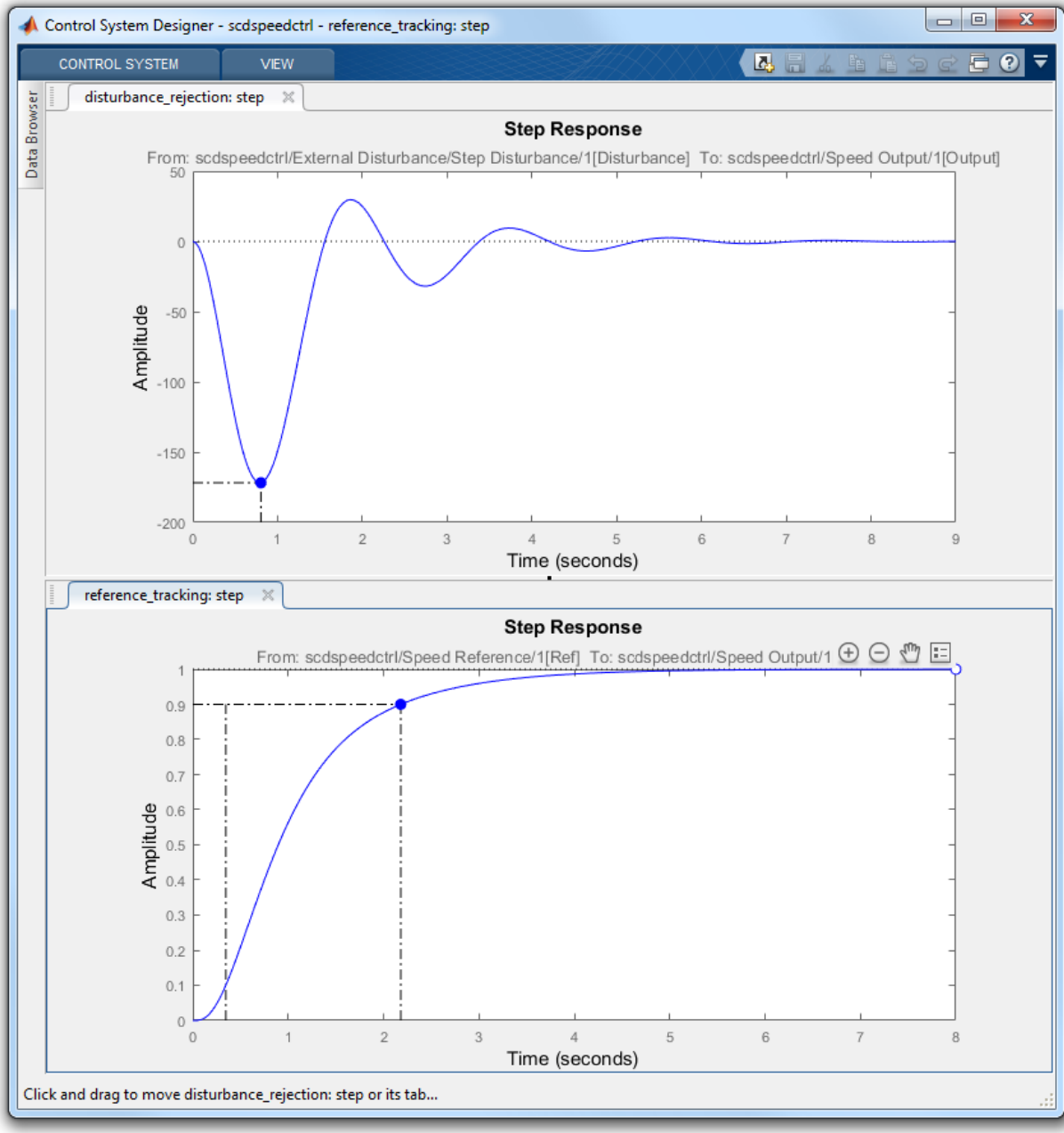
- `scdspeedctrl/PID Controller`

  ```
  P = 0.0012191

  I = 0.0030038
  ```

- `scdspeedctrl/Reference Filter`:

  ```
  Numerator = 10;

  Denominator = [1 10];
  ```

In the Compensator Editor dialog box, specify these parameters. The responses of the closed-loop system are shown below:


Control System Designer - scdspeedctrl - reference_tracking: step
CONTROL SYSTEM
VIEW
Data Browser
disturbance_rejection: step
Step Response
From: scdspeedctrl/External Disturbance/Step Disturbance/1[Disturbance] To: scdspeedctrl/Speed Output/1[Output]
Amplitude
Time (seconds)
reference_tracking: step
Step Response
From: scdspeedctrl/Speed Reference/1[Ref] To: scdspeedctrl/Speed Output/1
Amplitude
Time (seconds)
Click and drag to move disturbance_rejection: step or its tab...

Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdspeedctrl')
```

See Also

Control System Designer

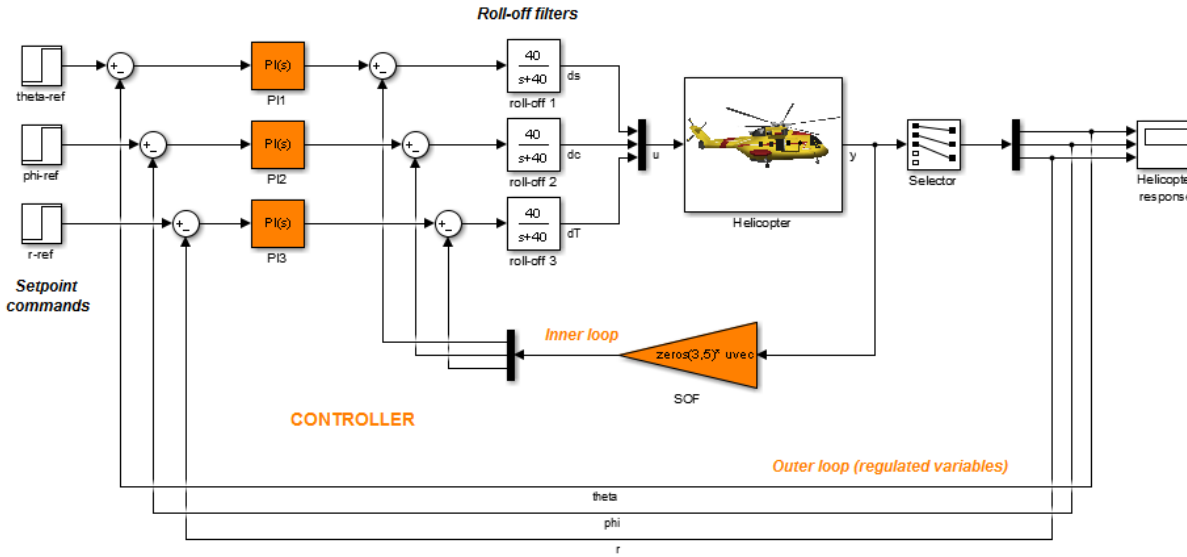
More About

- “Edit Compensator Dynamics” (Control System Toolbox)
- “Update Simulink Model and Validate Design” on page 8-51

Control System Tuning

Automated Tuning Overview

The control system tuning tools such as `systune` and Control System Tuner automatically tune control systems from high-level tuning goals you specify, such as reference tracking, disturbance rejection, and stability margins. The software jointly tunes all the free parameters of your control system regardless of control system architecture or the number of feedback loops it contains. For example, the model of the following illustration represents a multiloop control system for a helicopter.



This control system includes a number of fixed elements, such as the helicopter model itself and the roll-off filters. The inner control loop provides static output feedback for decoupling. The outer loop includes PI controllers for setpoint tracking. The tuning tools jointly optimize the gains in the `SOF` and `PI` blocks to meet setpoint tracking, stability margin, and other requirements that you specify. These tools allow you to specify any control structure and designate which blocks in your system are tunable.

Control systems are tuned to meet your specific performance and robustness goals subject to feasibility constraints such as actuator limits, sensor accuracy, computing power, or energy consumption. The library of tuning goals lets you capture these objectives in a form suitable for fast automated tuning. This library includes standard control objectives such as reference tracking, disturbance rejection, loop shapes, closed-

loop damping, and stability margins. Using these tools, you can perform multi-objective tuning of control systems having any structure.

See Also

Control System Designer | `systune`

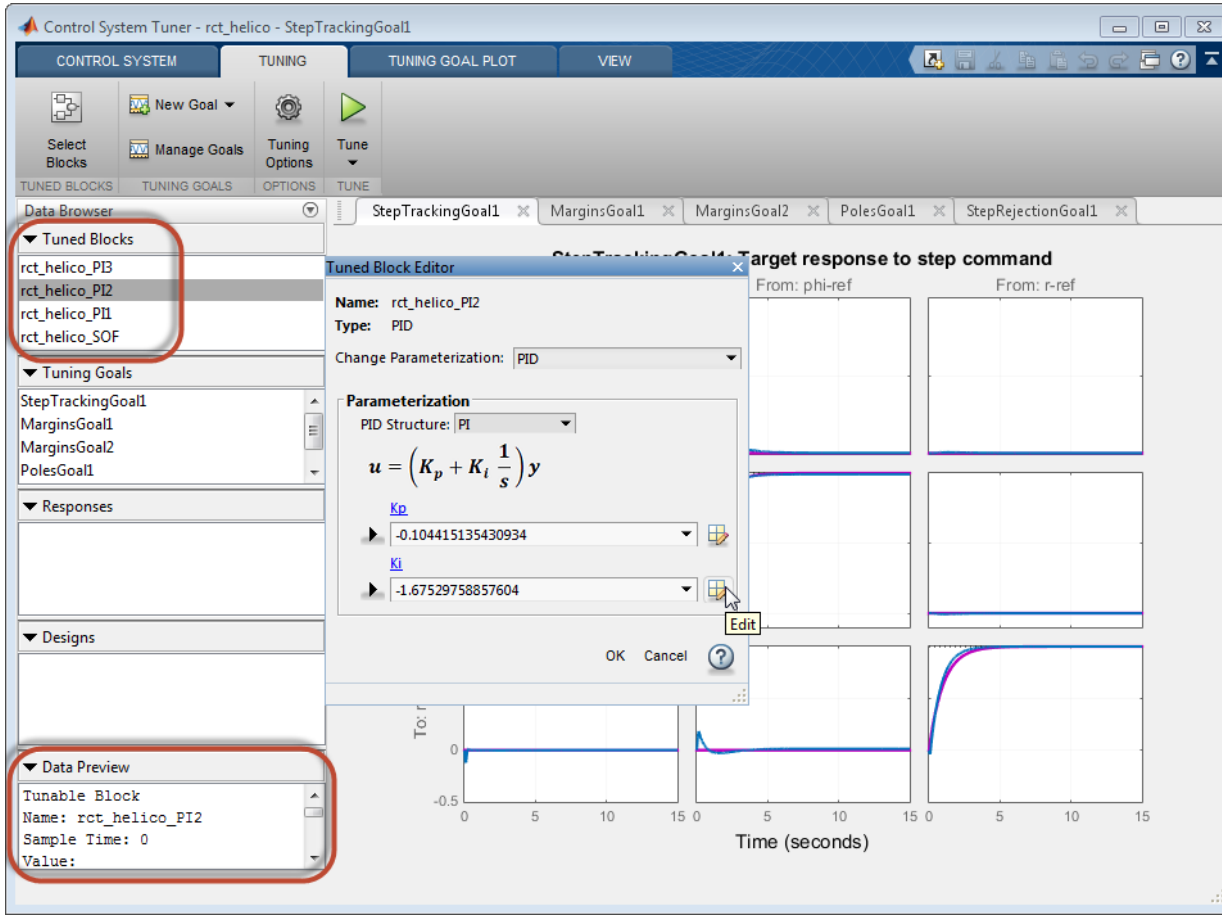
More About

- “Choosing an Automated Tuning Approach” on page 9-5
- “Automated Tuning Workflow” on page 9-7

Choosing an Automated Tuning Approach

You can tune control systems at the MATLAB command line or using the Control System Tuner App.

Control System Tuner provides an interactive graphical interface for specifying your tuning goals and validating the performance of the tuned control system.



Use Control System Tuner to tune control systems consisting of any number of feedback loops, with tunable components having any structure (such as PID, gain block, or state-space). You can represent your control architecture in MATLAB as a tunable generalized

state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model. Use the graphical interface to configure your tuning goals, examine response plots, and validate your controller design.

The `systune` command can perform all the same tuning tasks as Control System Tuner. Tuning at the command line allows you to write scripts for repeated tuning tasks. `systune` also provides advanced techniques such as tuning a controller for multiple plants, or designing gain-scheduled controllers. To use the command-line tuning tools, you can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model using an `slTuner` interface. Use the `TuningGoal` requirement objects to configure your tuning goals. Analysis commands such as `getIOTransfer` and `viewGoal` let you examine and validate the performance of your tuned system.

See Also

Control System Designer | `systune`

More About

- “Automated Tuning Workflow” on page 9-7

Automated Tuning Workflow

Whether you are tuning a control system at the command line or using Control System Tuner, the basic workflow includes the following steps:

1. Define your control architecture, by building a model of your control system from fixed-value blocks and blocks with tunable parameters. You can do so in one of several ways:

   - Create a Simulink model of your control system. (Tuning a Simulink model requires Simulink Control Design software.)
   - Use a predefined control architecture available in Control System Tuner.
   - At the command line, build a tunable `genss` model of your control system out of numeric LTI models and tunable control design blocks.

   For more information, see "Specify Control Architecture in Control System Tuner" on page 9-9.

2. Set up your model for tuning.

   - In Control System Tuner, identify which blocks of the model you want to tune. See Model Setup for Control System Tuner.
   - If tuning a Simulink model at the command line, create and configure the `slTuner` interface to the model. See Model Setup for Tuning at the Command Line.

3. Specify your tuning goals. Use the library of tuning goals to capture requirements such as reference tracking, disturbance rejection, stability margins, and more.

   - In Control System Tuner, use the graphical interface to specify tuning goals. See Tuning Goals (Control System Tuner).
   - At the command-line, use the `TuningGoal` requirement objects to specify your tuning goals. See Tuning Goals (programmatic tuning).

4. Tune the model. Use the `systune` command or Control System Tuner to optimize the tunable parameters of your control system to best meet your specified tuning goals. Then, analyze the tuned system responses and validate the design. Whether at the command line or in Control System Tuner, you can plot system responses to examine any aspects of system performance you need to validate your design.

   - For tuning and validating in Control System Tuner, see Tuning, Analysis, and Validation (Control System Tuner).

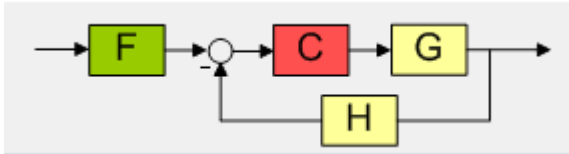
- For tuning at the command line, see Tuning, Analysis, and Validation (programmatic tuning).

Specify Control Architecture in Control System Tuner

In this section...

About Control Architecture

Control System Tuner lets you tune a control system having any architecture. Control system architecture defines how your controllers interact with the system under control. The architecture comprises the tunable control elements of your system, additional filter and sensor components, the system under control, and the interconnections among all these elements. For example, a common control system architecture is the single-loop feedback configuration of the following illustration:



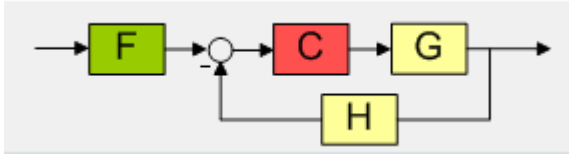
G is the plant model, and *H* the sensor dynamics. These are usually the fixed components of the control system. The prefilter *F* and feedback controller *C* are the tunable elements. Because control systems are so conveniently expressed in this block diagram form, these elements are referred to as fixed blocks and tunable blocks.

Control System Tuner gives you several ways to define your control system architecture:

- Use the predefined feedback structure of the illustration.
- Model any control system architecture in MATLAB by building a generalized state-space (`genss`) model from fixed LTI components and tunable control design blocks.
- Model your control system in Simulink and specify the blocks to tune in Control System Tuner (requires Simulink Control Design software).

Predefined Feedback Architecture

If your control system has the single-loop feedback configuration of the following illustration, use the predefined feedback structure built into Control System Tuner.



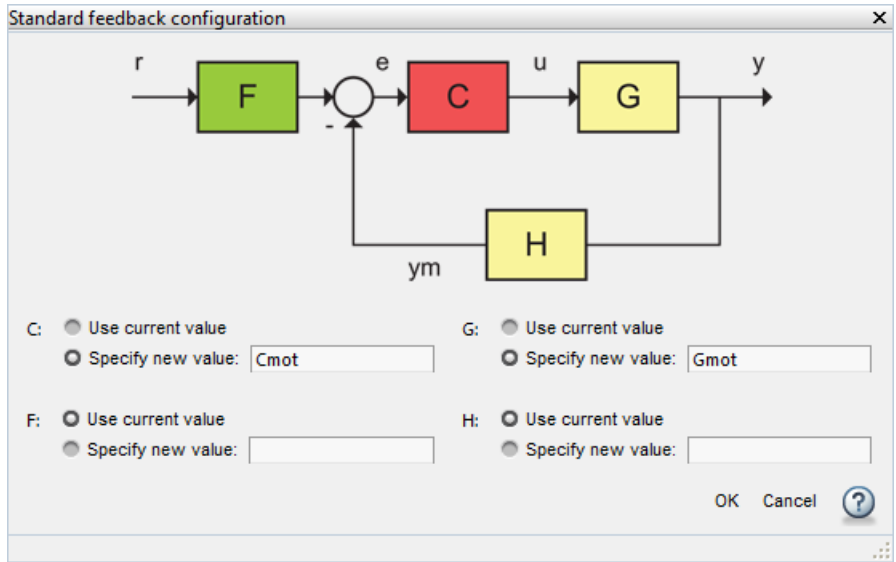
For example, suppose you have a DC motor for which you want to tune a PID controller. The response of the motor is modeled as $G(s) = 1/(s + 1)^2$. Create a fixed LTI model representing the plant, and a tunable PID controller model.

```
Gmot = zpk([],[-1,-1],1);
Cmot = tunablePID('Cmot','PID');
```

Open Control System Tuner.

```
controlSystemTuner
```

Control System Tuner opens, set to tune this default architecture. Next, specify the values of the blocks in the architecture. Click  to open the **Standard feedback configuration** dialog box.



Enter the values for *C* and *G* that you created. Control System Tuner reads these values from the MATLAB workspace. Click **OK**.

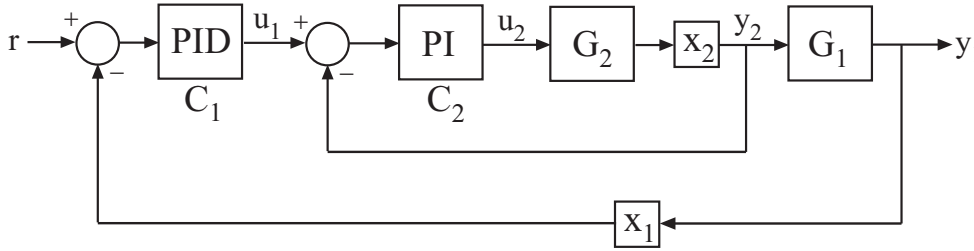
The default value for the sensor dynamics is a fixed unity-gain transfer function. The default value for the filter *F* is a tunable gain block.

You can now select blocks to tune, create tuning goals, and tune the control system.

Arbitrary Feedback Control Architecture

If your control architecture does not match Control System Tuner's predefined control architecture, you can create a generalized state-space (`genss`) model with tunable components representing your controller elements. For example, suppose you want to tune the cascaded control system of the following illustration, that includes two tunable PID controllers.

.



Create tunable control design blocks for the controllers, and fixed LTI models for the plant components, G_1 and G_2. Also include optional loop-opening locations x_1 and x_2. These locations indicate where you can open loops or inject signals for the purpose of specifying requirements for tuning the system.

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);

C20 = tunablePID('C2','pi');
C10 = tunablePID('C1','pid');

X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CL0 = feedback(G1*InnerLoop*C10,X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

`CL0` is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Open Control System Tuner to tune this model.

```
controlSystemTuner(CL0)
```

You can now select blocks to tune, create tuning goals, and tune the control system.

Related Examples

- “Building Tunable Models” (Control System Toolbox)
- “Specify Blocks to Tune in Control System Tuner” on page 9-25

- "Specify Goals for Interactive Tuning" on page 9-40

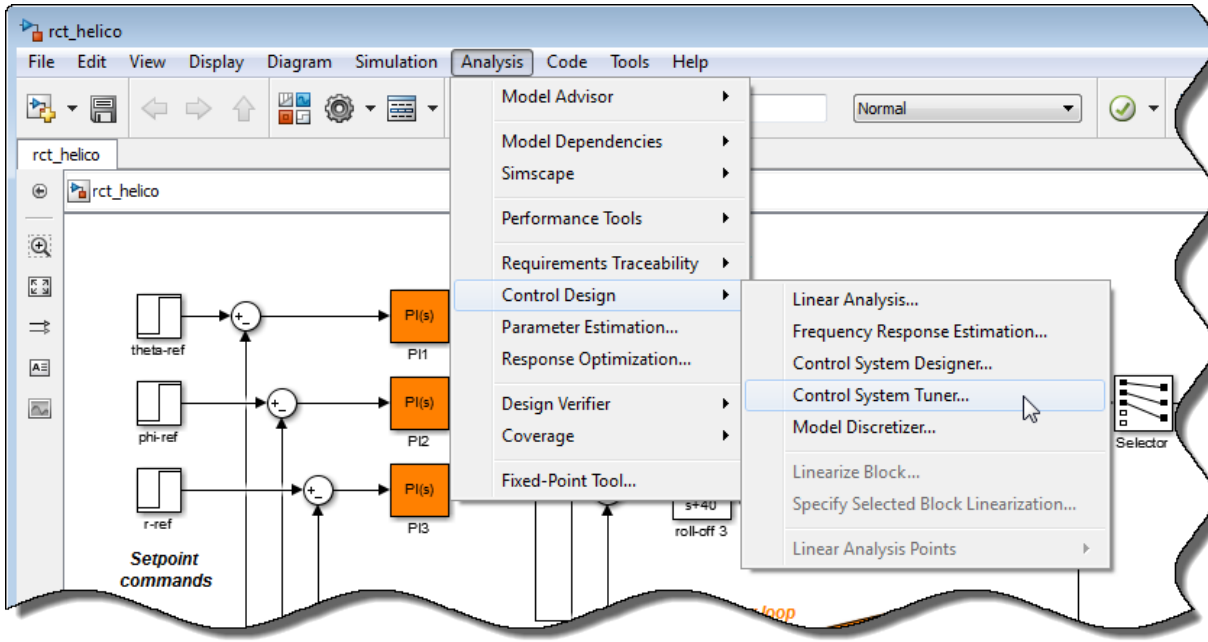
Control System Architecture in Simulink

If you have Simulink Control Design software, you can model an arbitrary control system architecture in a Simulink model and tune the model in Control System Tuner.

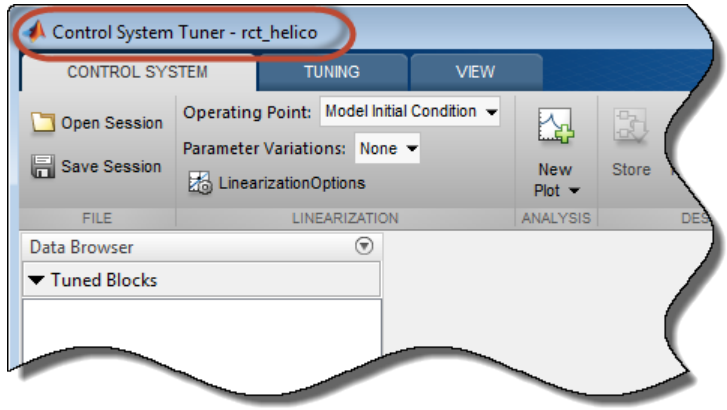
See "Open Control System Tuner for Tuning Simulink Model" on page 9-14.

Open Control System Tuner for Tuning Simulink Model

To open Control System Tuner for tuning a Simulink model, open the model. In the Simulink Editor, select **Analysis > Control Design > Control System Tuner**.



Each instance of Control System Tuner is linked to the Simulink model from which it is opened. The title bar of the Control System Tuner window reflects the name of the associated Simulink model.



Command-Line Equivalents

At the MATLAB command line, use the `controlSystemTuner` command to open Control System Tuner for tuning a Simulink model. For example, the following command opens Control System Tuner for the model `rct_helico.slx`.

```
controlSystemTuner('rct_helico')
```

If `SLT0` is an `slTuner` interface to the Simulink model, the following command opens Control System Tuner using the information in the interface, such as blocks to tune and analysis points.

```
controlSystemTuner(SLT0)
```

See Also

Related Examples

- "Specify Operating Points for Tuning in Control System Tuner" on page 9-16
- "Specify Blocks to Tune in Control System Tuner" on page 9-25

More About

- "Automated Tuning Workflow" on page 9-7

Specify Operating Points for Tuning in Control System Tuner

In this section...

About Operating Points in Control System Tuner

When you use Control System Tuner with a Simulink model, the software computes system responses and tunes controller parameters for a linearization of the model. That linearization can depend on model operating conditions.

By default, Control System Tuner linearizes at the operating point specified in the model, which comprises the initial state values in the model (the model initial conditions). You can specify one or more alternate operating points for tuning the model. Control System Tuner lets you compute two types of alternate operating points:

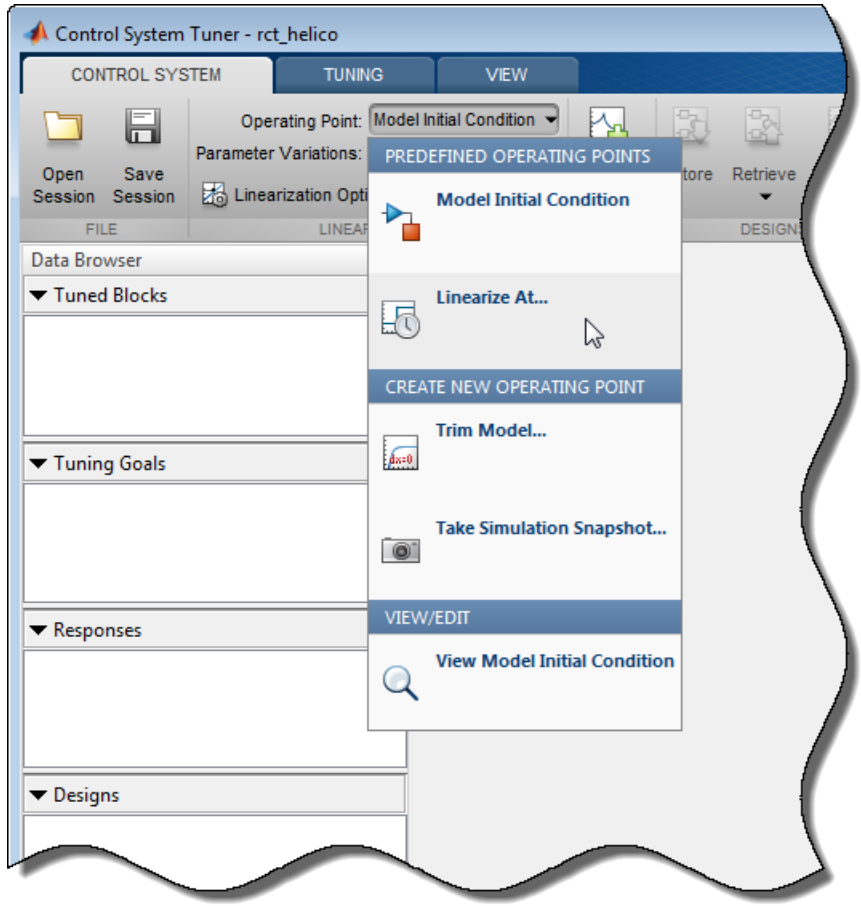
- Simulation snapshot time. Control System Tuner simulates the model for the amount of time you specify, and linearizes using the state values at that time. Simulation snapshot linearization is useful, for instance, when you know your model reaches an equilibrium state after a certain simulation time.
- Steady-state operating point. Control System Tuner finds a steady-state operating point at which some specified condition is met (trimming). For example, if your model represents an automobile motor, you can compute an operating point at which the motor operates steadily at 2000 rpm.

For more information on finding steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-6.

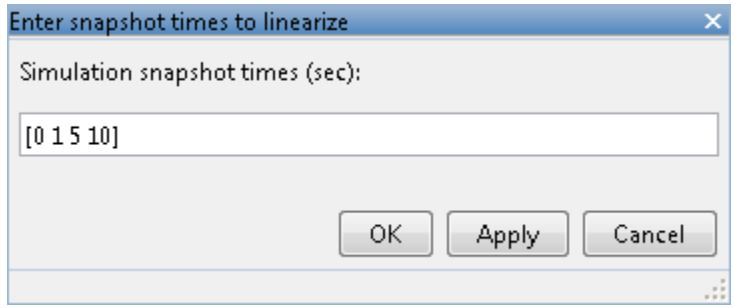
Linearize at Simulation Snapshot Times

This example shows how to compute linearizations at one or more simulation snapshot times.

In the **Control System** tab, in the **Operating Point** menu, select `Linearize At`.



In the **Enter snapshot times to linearize** dialog box, specify one or more simulation snapshot times. Click **OK**.

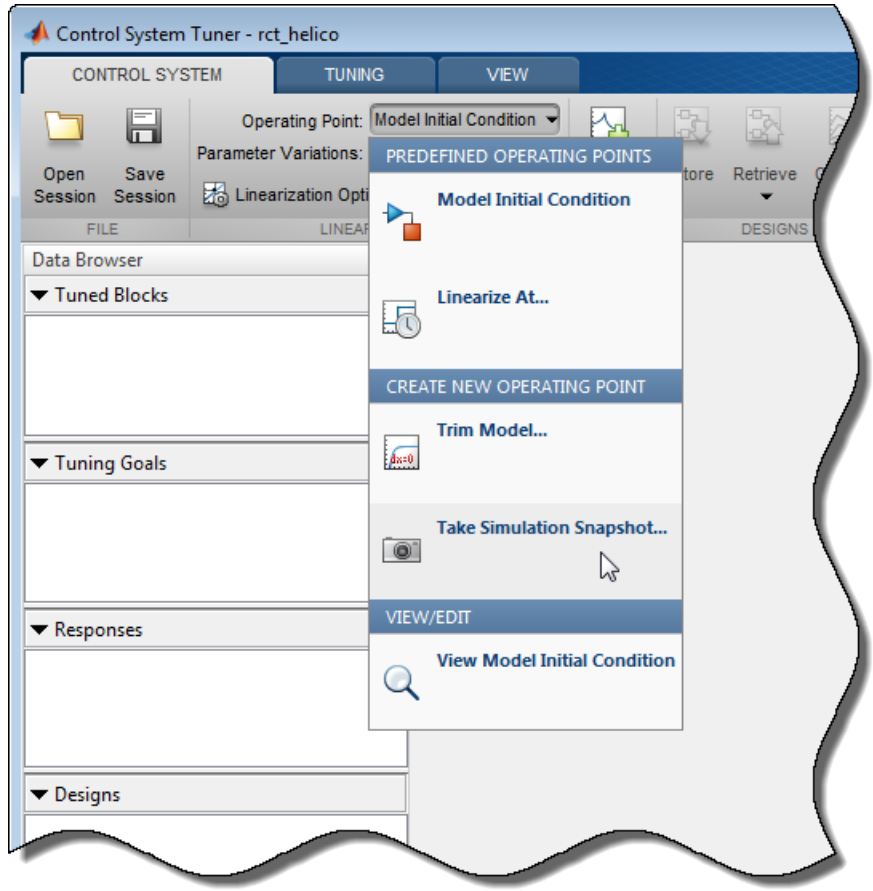


When you are ready to analyze system responses or tune your model, Control System Tuner computes linearizations at the specified snapshot times. If you enter multiple snapshot times, Control System Tuner computes an array of linearized models, and displays analysis plots that reflect the multiple linearizations in the array. In this case, Control System Tuner also takes into account all linearizations when tuning parameters. This helps to ensure that your tuned controller meets your design requirements at a variety of operating conditions.

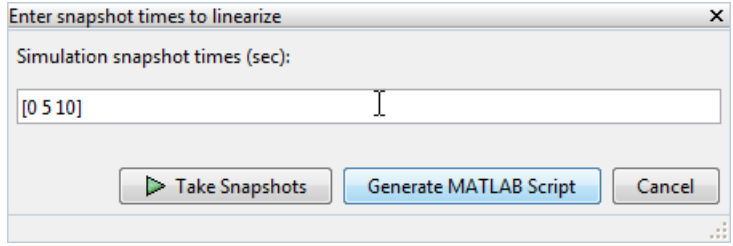
Compute Operating Points at Simulation Snapshot Times

This example shows how to compute operating points at one or more simulation snapshot times. Doing so stores the operating point within Control System Tuner. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Take simulation snapshot`.



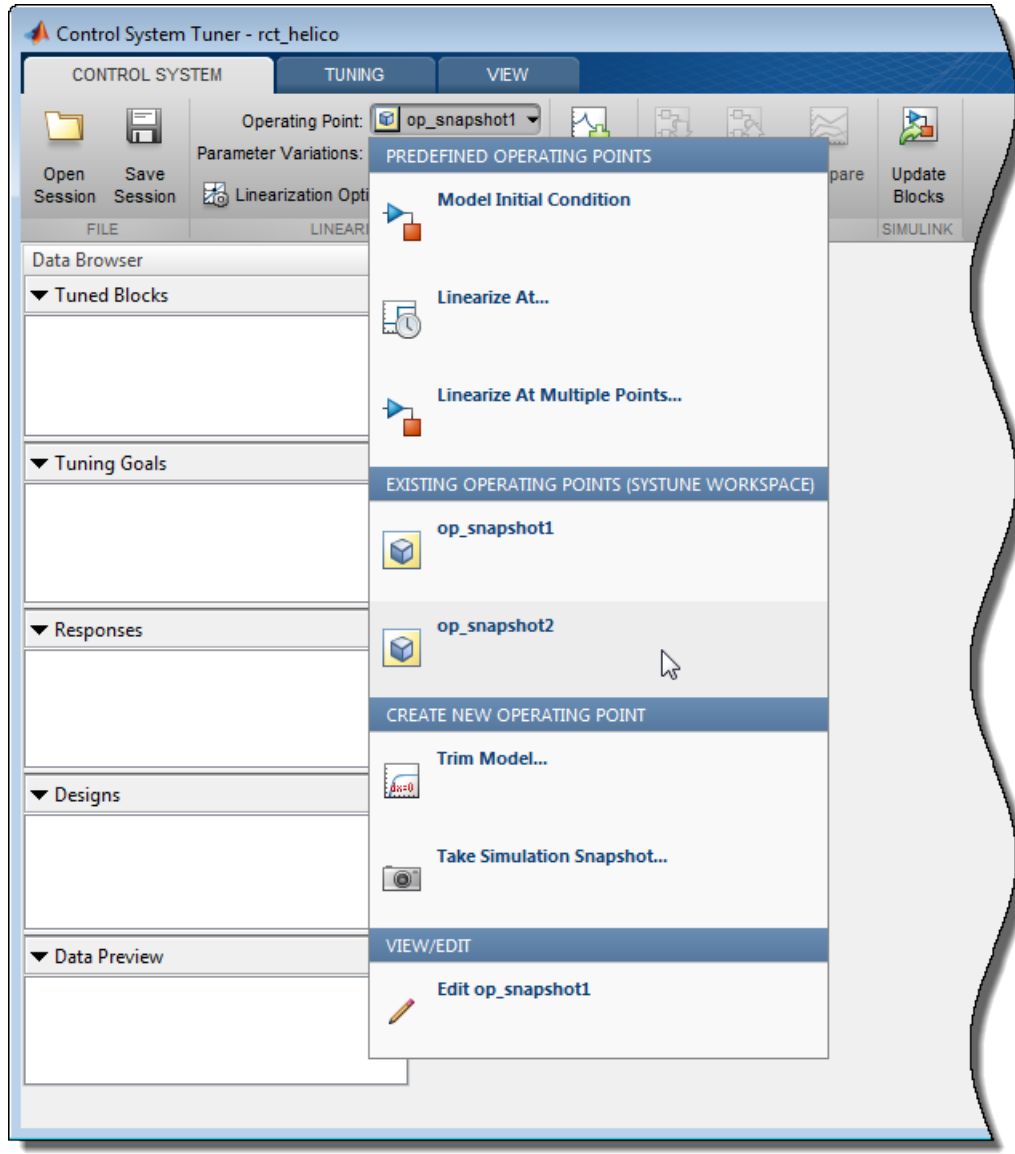
In the **Enter snapshot times to linearize** dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



Click  **Take Snapshots**. Control System Tuner simulates the model and computes the snapshot operating points.

Compute additional snapshot operating points if desired. Enter additional snapshot times and click  **Take Snapshots**. Close the dialog box when you are done.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.

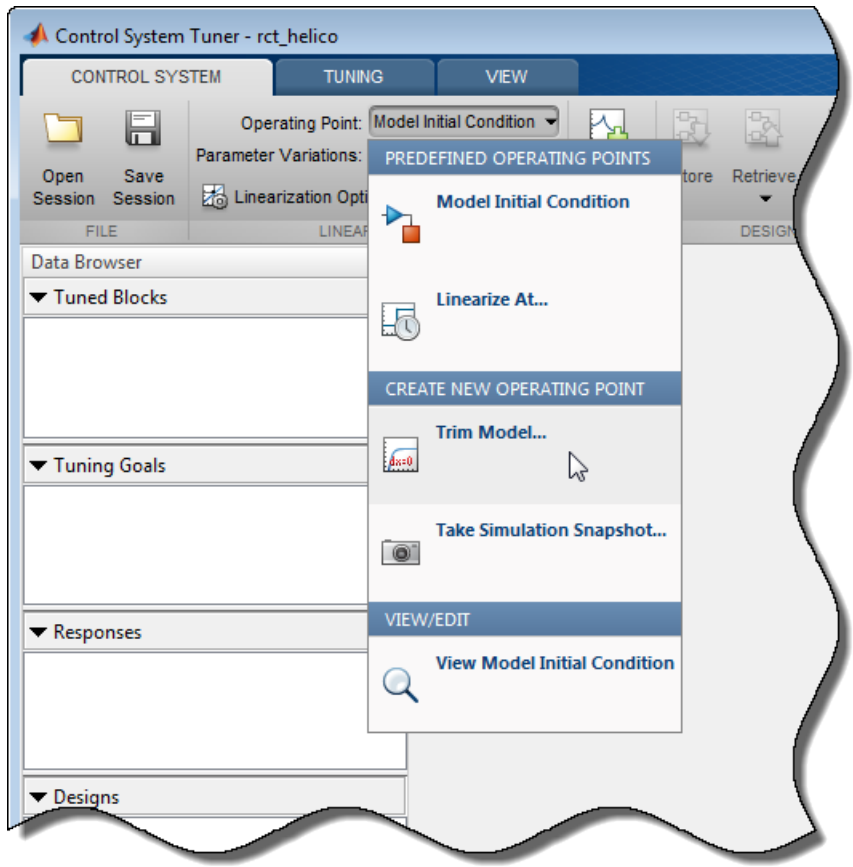

Control System Tuner - rct_helico
CONTROL SYSTEM
TUNING
VIEW
Operating Point:
op_snapshot1
Parameter Variations:
Linearization Opti
Open Session
Save Session
FILE
Update Blocks
SIMULINK
Data Browser
Tuned Blocks
Tuning Goals
Responses
Designs
Data Preview
PREDEFINED OPERATING POINTS
Model Initial Condition
Linearize At...
Linearize At Multiple Points...
EXISTING OPERATING POINTS (SYSTUNE WORKSPACE)
op_snapshot1
op_snapshot2
CREATE NEW OPERATING POINT
Trim Model...
Take Simulation Snapshot...
VIEW/EDIT
Edit op_snapshot1

If you entered a vector of snapshot times, all the resulting operating points are stored together in an operating-point vector. You can use this vector to tune a control system at several operating points simultaneously.

Compute Steady-State Operating Points

This example shows how to compute a steady-state operating point with specified conditions. Doing so stores the operating point within Control System Tuner. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Trim model`.

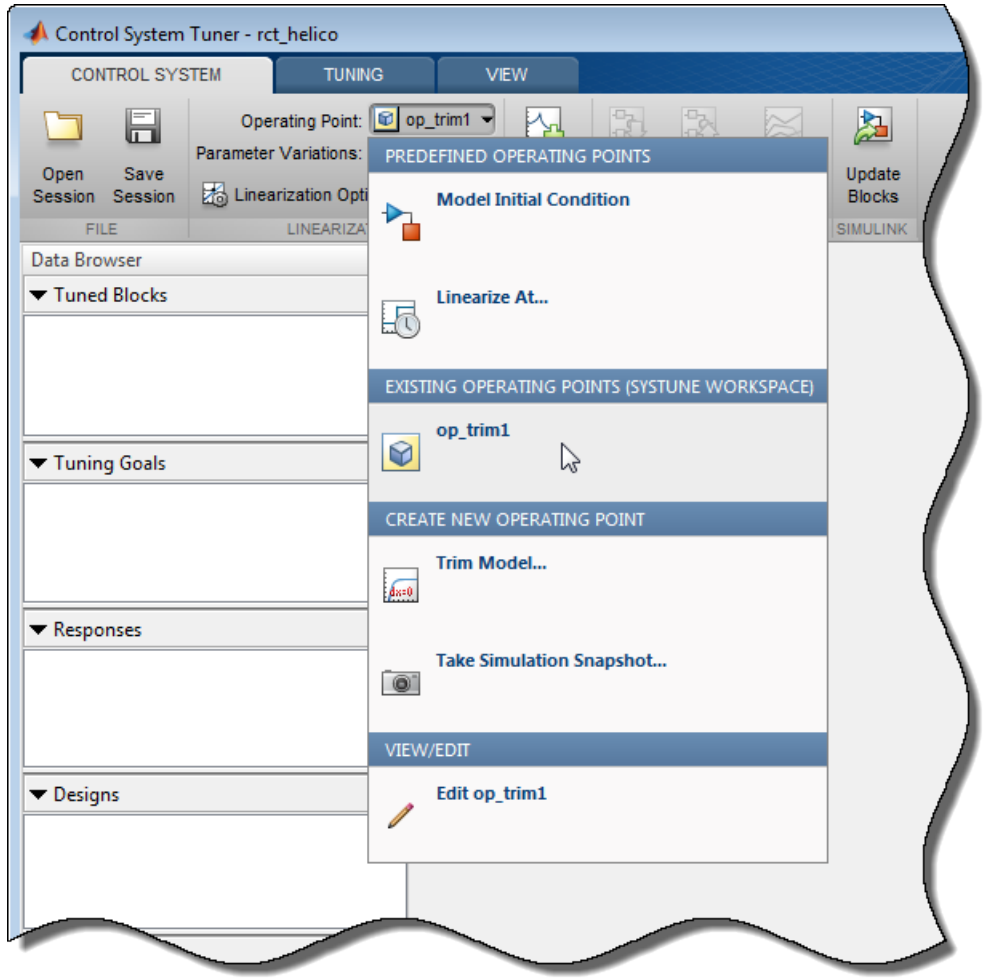


In the **Trim the model** dialog box, enter the specifications for the steady-state state values at which you want to find an operating point.

For examples showing how to use the **Trim the model** dialog box to specify the conditions for a steady-state operating point search, see "Compute Steady-State Operating Point from State Specifications" on page 1-14 and "Compute Steady-State Operating Point from Output Specifications" on page 1-28.

When you have entered your state specifications, click  **Start trimming**. Control System Tuner finds an operating point that meets the state specifications and stores it.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.



See Also

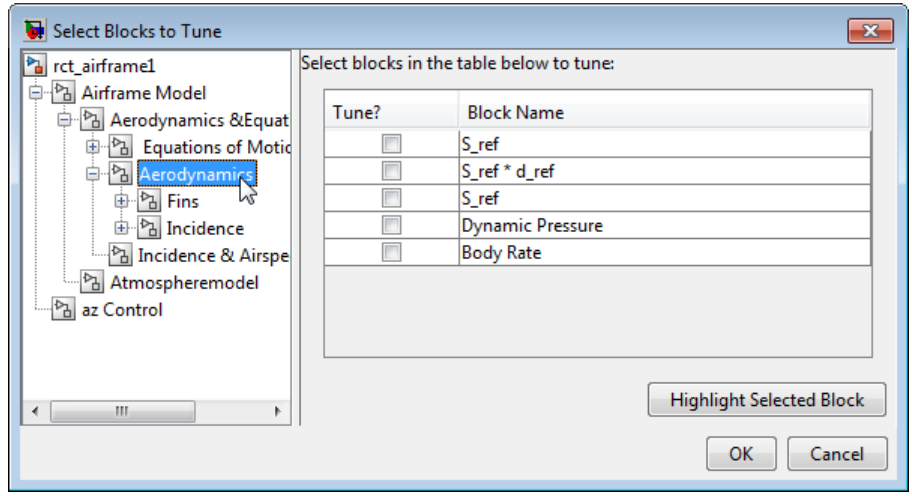
Related Examples

- “Specify Blocks to Tune in Control System Tuner” on page 9-25
- “Robust Tuning Approaches” (Robust Control Toolbox)

Specify Blocks to Tune in Control System Tuner

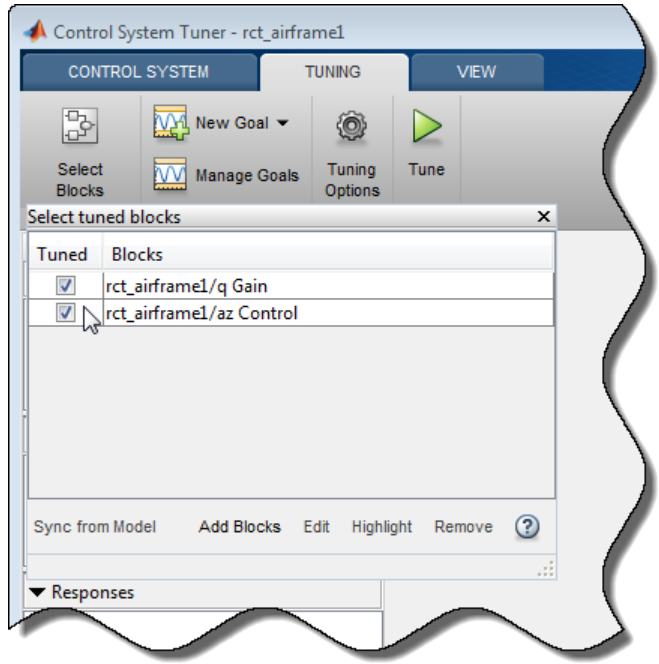
To select which blocks of your Simulink model to tune in Control System Tuner:

1. In the **Tuning** tab, click **Select Blocks**. The **Select tuned Blocks** dialog opens.
2. Click **Add Blocks**. Control System Tuner analyzes your model to find blocks that can be tuned.
3. In the **Select Blocks to Tune** dialog box, use the nodes in the left panel to navigate through your model structure to the subsystem that contains blocks you want to tune. Check **Tune?** for the blocks you want to tune. The parameters of blocks you do not check remain constant when you tune the model.



Tip To find a block in your model, select the block in the **Block Name** list and click **Highlight Selected Block**.

4. Click **OK**. The **Select tuned blocks** dialog box now reflects the blocks you added.



To import the current value of a block from your model into the current design in Control System Tuner, select the block in the **Blocks** list and click **Sync from Model**. Doing so is useful when you have tuned a block in Control System Tuner, but wish to restore that block to its original value. To store the current design before restoring a block value, in the **Control System** tab, click **Store**.

See Also

Related Examples

- “View and Change Block Parameterization in Control System Tuner” on page 9-27

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 9-37

View and Change Block Parameterization in Control System Tuner

Control System Tuner parameterizes every block that you designate for tuning.

- When you tune a Simulink model, Control System Tuner automatically assigns a default parameterization to tunable blocks in the model. The default parameterization depends on the type of block. For example, a PID Controller block configured for PI structure is parameterized by proportional gain and integral gain as follows:

  $$u = K_p + K_i \frac{1}{s}.$$

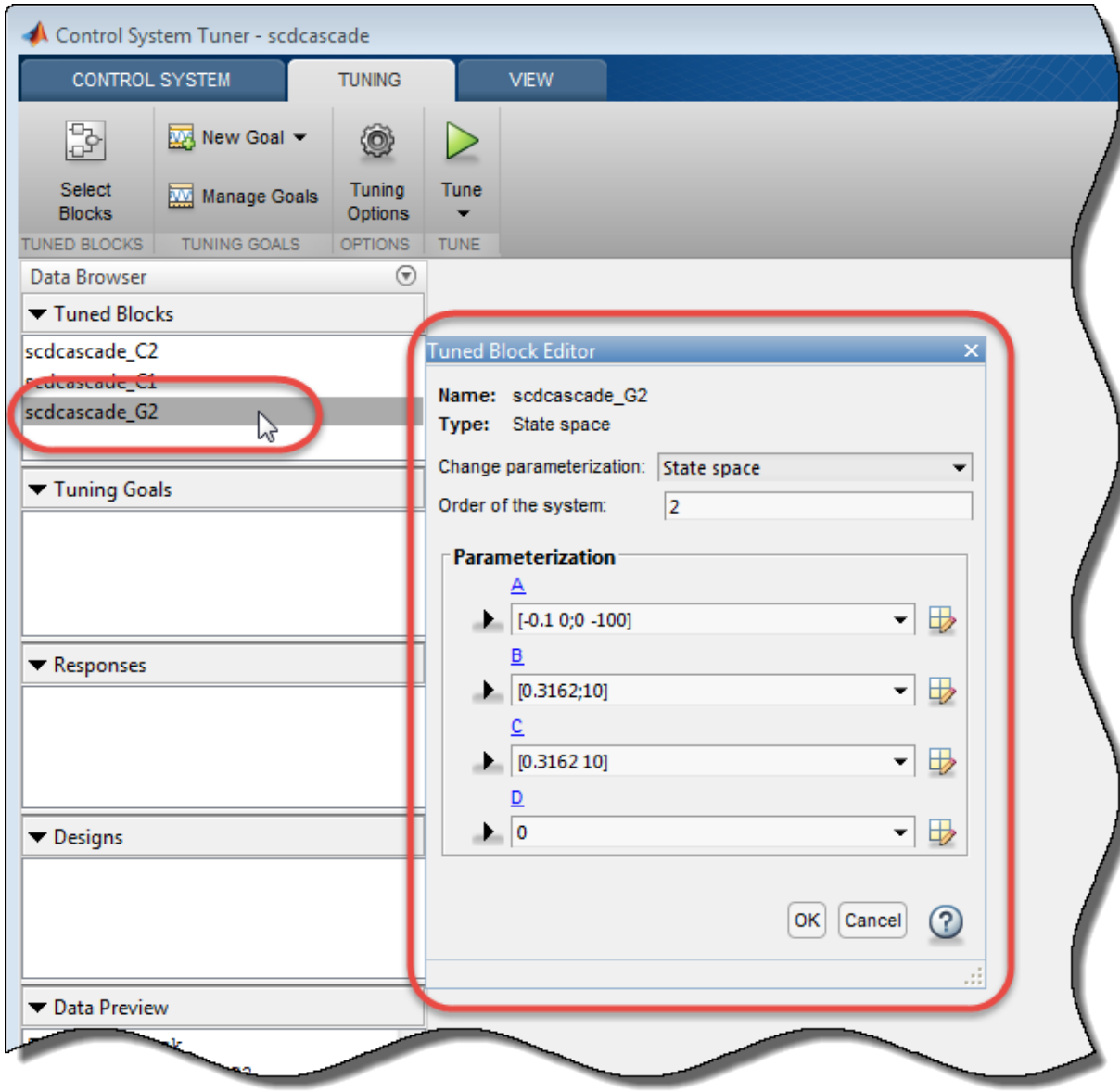
  K_p and K_i are the tunable parameters whose values are optimized to satisfy your specified tuning goals.
- When you tune a predefined control architecture or a MATLAB (generalized state-space) model, you define the parameterization of each tunable block when you create it at the MATLAB command line. For example, you can use `tunablePID` to create a tunable PID block.

Control System Tuner lets you view and change the parameterization of any block to be tuned. Changing the parameterization can include changing the structure or current parameter values. You can also designate individual block parameters fixed (non-tunable) or limit their tuning range.

In this section...
“View Block Parameterization” on page 9-27
“Fix Parameter Values or Limit Tuning Range” on page 9-29
“Custom Parameterization” on page 9-31
“Block Rate Conversion” on page 9-32

View Block Parameterization

To access the parameterization of a block that you have designated as a tuned block, in the **Data Browser**, in the **Tuned Blocks** area, double-click the name of a block. The Tuned Block Editor dialog box opens, displaying the current block parameterization.


Control System Tuner - scdcascade
CONTROL SYSTEM
TUNING
VIEW
Select Blocks
New Goal
Manage Goals
Tuning Options
Tune
TUNED BLOCKS
TUNING GOALS
OPTIONS
TUNE
Data Browser
Tuned Blocks
scdcascade_C2
scdcascade_G2
Tuning Goals
Responses
Designs
Data Preview
Tuned Block Editor
Name: scdcascade_G2
Type: State space
Change parameterization: State space
Order of the system: 2
Parameterization
A
[-0.1 0;0 -100]
B
[0.3162;10]
C
[0.3162 10]
D
0
OK
Cancel

The fields of the **Tuned Block Editor** display the type of parameterization, such as `PID`, `State-Space`, or `Gain`. For more specific information about the fields, click  .

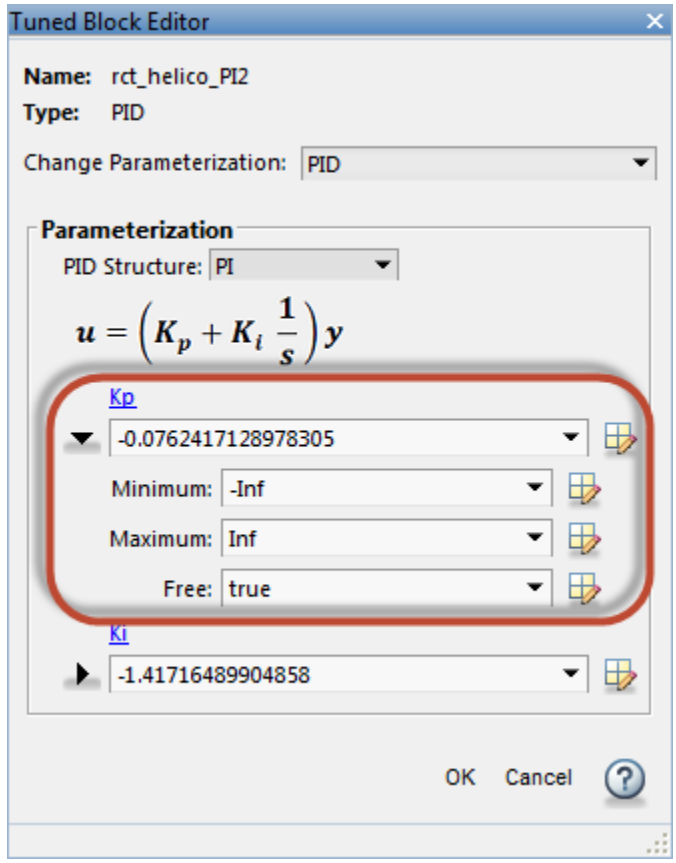
Note To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

Fix Parameter Values or Limit Tuning Range

You can change the current value of a parameter, fix its current value (make the parameter nontunable), or limit the parameter's tuning range.

To change a current parameter value, type a new value in its text box. Alternatively, click  to use a variable editor to change the current value. If you attempt to enter an invalid value, the parameter returns to its previous value.

Click  to access and edit additional properties of each parameter.



- **Minimum** — Minimum value that the parameter can take when the control system is tuned.
- **Maximum** — Maximum value that the parameter can take when the control system is tuned.
- **Free** — When the value is `true`, Control System Toolbox tunes the parameter. To fix the value of the parameter, set **Free** to `false`.

For array-valued parameters, you can set these properties independently for each entry in the array. For example, for a vector-valued gain of length 3, enter `[1 10 100]` to set the current value of the three gains to 1, 10, and 100 respectively. Alternatively, click  to use a variable editor to specify such values.

For vector or matrix-valued parameters, you can use the **Free** parameter to constrain the structure of the parameter. For example, to restrict a matrix-valued parameter to be a diagonal matrix, set the current values of the off-diagonal elements to 0, and set the corresponding entries in **Free** to `false`.

Custom Parameterization

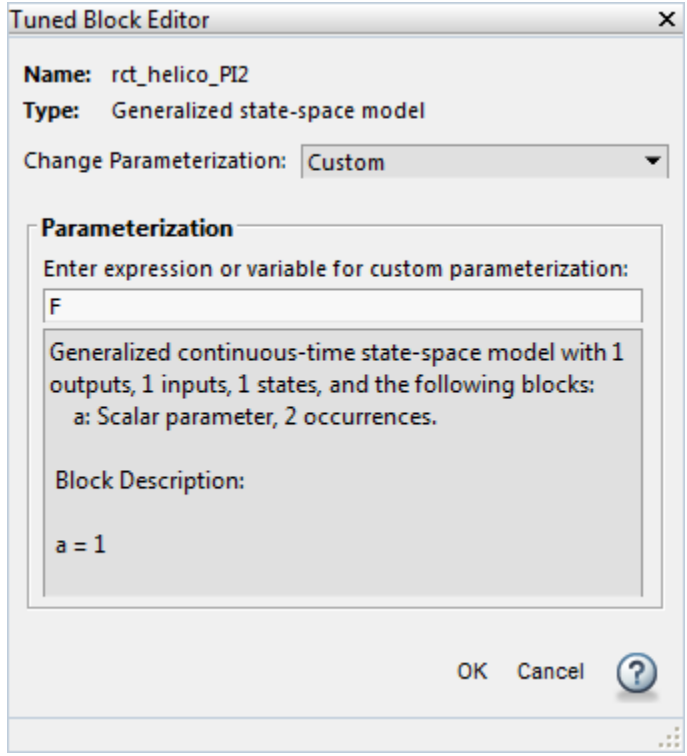
When tuning a control system represented by a Simulink model or by a "Predefined Feedback Architecture" on page 9-10, you can specify a custom parameterization for any tuned block using a generalized state-space (`genss`) model. To do so, create and configure a `genss` model in the MATLAB workspace that has the desired parameterization, initial parameter values, and parameter properties. In the **Change parameterization** dialog box, select `Custom`. In the **Parameterization** area, the variable name of the `genss` model.

For example, suppose you want to specify a tunable low-pass filter, $F = a/(s+a)$, where a is the tunable parameter. First, at the MATLAB command line, create a tunable `genss` model that represents the low-pass filter structure.

```
a = realp('a',1);
F = tf(a,[1 a]);
```

```
F =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs,
  1 states, and the following blocks:
    a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value, "get(F)" to see all properties, and
"F.Blocks" to interact with the blocks.
```

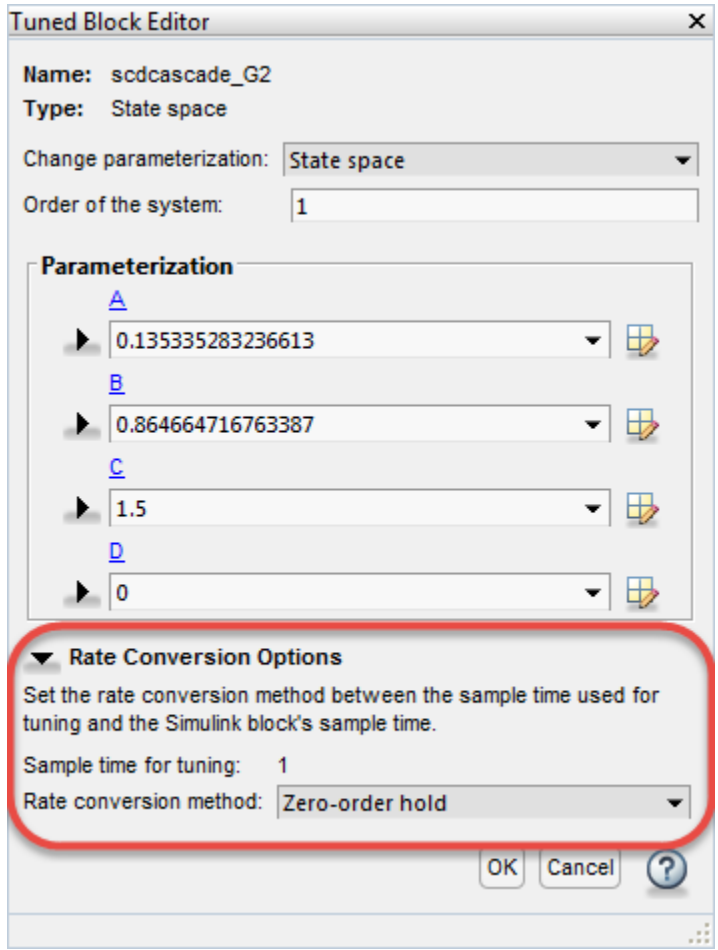
Then, in the Tuned Block Editor, enter `F` in the **Parameterization** area.



When you specify a custom parameterization for a Simulink block, you might not be able to write the tuned block value back to the Simulink model. When writing values to Simulink blocks, Control System Tuner skips blocks that cannot represent the tuned value in a straightforward and lossless manner. For example, if you reparameterize a PID Controller Simulink block as a third-order state-space model, Control System Tuner will not write the tuned value back to the block.

Block Rate Conversion

When Control System Tuner writes tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. When the two sample times differ, the Tuned Block Editor contains additional rate conversion options that specify how this resampling operation is performed for the corresponding block.



By default, Control System Tuner performs linearization and tuning in continuous time (sample time = 0). You can specify discrete-time linearization and tuning and change the sample time. To do so, on the **Control System** tab, click **Linearization Options**. **Sample time for tuning** reflects the sample time specified in the **Linearization Options** dialog box.

The remaining rate conversion options depend on the parameterized block.

Rate Conversion for Parameterized PID Blocks

For parameterization of continuous-time PID Controller and PID Controller (2-DOF) blocks, you can independently specify the rate-conversion methods as discretization formulas for the integrator and derivative filter. Each has the following options:

- `Trapezoidal` (default) — Integrator or derivative filter discretized as `(Ts/2)*(z+1)/(z-1)`, where `Ts` is the target sample time.
- `Forward Euler` — `Ts/(z-1)`.
- `Backward Euler` — `Ts*z/(z-1)`.

For more information about PID discretization formulas, see "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" (Control System Toolbox).

For discrete-time PID Controller and PID Controller (2-DOF) blocks, you set the integrator and derivative filter methods in the block dialog box. You cannot change them in the Tuned Block Editor.

Rate Conversion for Other Parameterized Blocks

For blocks other than PID Controller blocks, the following rate-conversion methods are available:

- `Zero-order hold` — Zero-order hold on the inputs. For most dynamic blocks this is the default rate-conversion method.
- `Tustin` — Bilinear (Tustin) approximation.
- `Tustin with prewarping` — Tustin approximation with better matching between the original and rate-converted dynamics at the prewarp frequency. Enter the frequency in the **Prewarping frequency** field.
- `First-order hold` — Linear interpolation of inputs.
- `Matched` (SISO only) — Zero-pole matching equivalents.

For more detailed information about these rate-conversion methods, see "Continuous-Discrete Conversion Methods" (Control System Toolbox).

Blocks with Fixed Rate Conversion Methods

For the following blocks, you cannot set the rate-conversion method in the Tuned Block Editor.

- Discrete-time PID Controller and PID Controller (2-DOF) block. Set the integrator and derivative filter methods in the block dialog box.
- Gain block, because it is static.
- Transfer Fcn Real Zero block. This block can only be tuned at the sample time specified in the block.
- Block that has been discretized using the Model Discretizer. Sample time for this block is specified in the Model Discretizer itself.

See Also

Related Examples

- “Specify Blocks to Tune in Control System Tuner” on page 9-25

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 9-37

Setup for Tuning Control System Modeled in MATLAB

To model your control architecture in MATLAB for tuning in Control System Tuner, construct a tunable model of the control system that identifies and parameterizes its tunable elements. You do so by combining numeric LTI models of the fixed elements with parametric models of the tunable elements. The result is a tunable generalized state-space `genss` model.

Building a tunable `genss` model for Control System Tuner is the same as building such a model for tuning at the command line. For information about building such models, "Setup for Tuning MATLAB Models" (Control System Toolbox).

When you have a tunable `genss` model of your control system, use the `controlSystemTuner` command to open Control System Tuner. For example, if `T0` is the `genss` model, the following command opens Control System Tuner for tuning `T0`:

```
controlSystemTuner(T0)
```

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40

How Tuned Simulink Blocks Are Parameterized

Blocks With Predefined Parameterization

When you tune a Simulink model, either with Control System Tuner or at the command line through an `slTuner` interface, the software automatically assigns a predefined parameterization to certain Simulink blocks. For example, for a PID Controller block set to the PI controller type, the software automatically assigns the parameterization $K_p + K_i/s$, where K_p and K_i are the tunable parameters. For blocks that have a predefined parameterization, you can write tuned values back to the Simulink model for validating the tuned controller.

Blocks that have a predefined parameterization include the following:

Simulink Library	Blocks with Predefined Parameterization
Math Operations	Gain
Continuous	• State-Space • Transfer Fcn • Zero-Pole • PID Controller • PID Controller (2 DOF)
Discrete	• Discrete State-Space • Discrete Transfer Fcn • Discrete Zero-Pole • Discrete Filter • Discrete PID Controller • Discrete PID Controller (2 DOF)
Lookup Tables	• 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table
Control System Toolbox	LTI System

Simulink Library	Blocks with Predefined Parameterization
Discretizing (Model Discretizer Blocks)	• Discretized State-Space • Discretized Transfer Fcn • Discretized Zero-Pole • Discretized LTI System • Discretized Transfer Fcn (with initial states)
Simulink Extras/Additional Linear	State-Space (with initial outputs)

Scalar Expansion

The following tunable blocks support scalar expansion:

- Discrete Filter
- Gain
- 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table
- PID Controller, PID Controller (2DOF)

Scalar expansion means that the block parameters can be scalar values even when the input and output signals are vectors. For example, you can use a Gain block to implement `y = k*u` with scalar `k` and vector `u` and `y`. To do so, you set the **Multiplication** mode of the block to `Element-wise(K.*u)`, and set the gain value to the scalar `k`.

When a tunable block uses scalar expansion, its default parameterization uses tunable scalars. For example, in the `y = k*u` Gain block, the software parameterizes the scalar `k` as a tunable real scalar (`realp` of size `[1 1]`). If instead you want to tune different gain values for each channel, replace the scalar gain `k` by a `N-by-1` gain vector in the block dialog, where `N` is the number of channels, the length of the vectors `u` and `y`. The software then parameterizes the gain as a `realp` of size `[N 1]`.

Blocks Without Predefined Parameterization

You can specify blocks for tuning that do not have a predefined parameterization. When you do so, the software assigns a state-space parameterization to such blocks based upon the block linearization. For blocks that do not have a predefined parameterization, the software cannot write tuned values back to the block, because there is no clear mapping

between the tuned parameters and the block. To validate a tuned control system that contains such blocks, you can specify a block linearization in your model using the value of the tuned parameterization. (See “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-158 for more information about specifying block linearization.)

View and Change Block Parameterization

You can view and edit the current parameterization of every block you designate for tuning.

- In Control System Tuner, see “View and Change Block Parameterization in Control System Tuner” on page 9-27.
- At the command line, use `getBlockParam` to view the current block parameterization. Use `setBlockParam` to change the block parameterization.

Specify Goals for Interactive Tuning

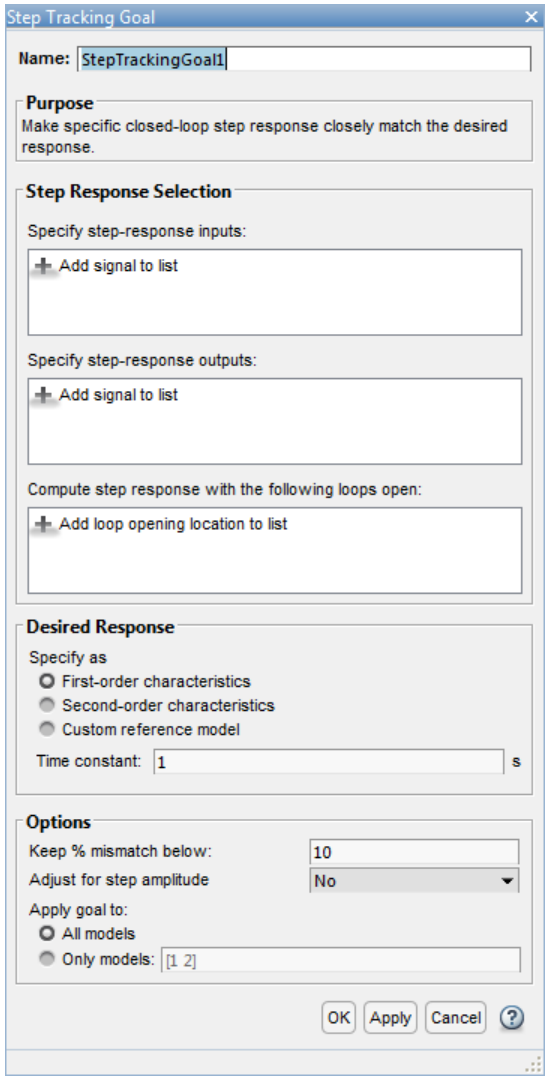
This example shows how to specify your tuning goals for automated tuning in Control System Tuner.

Use the **New Goal** menu to create a tuning goal such as a tracking requirement, disturbance rejection specification, or minimum stability margins. Then, when you are ready to tune your control system, use **Manage Goals** to designate which goals to enforce.

This example creates tuning goals for tuning the sample model `rct_helico`.

Choose Tuning Goal Type

In Control System Tuner, in the **Tuning** tab, click  **New Goal**. Select the type of goal you want to create. A tuning goal dialog box opens in which you can provide the detailed specifications of your goal. For example, select **Tracking of step commands** to make a particular step response of your control system match a desired response.



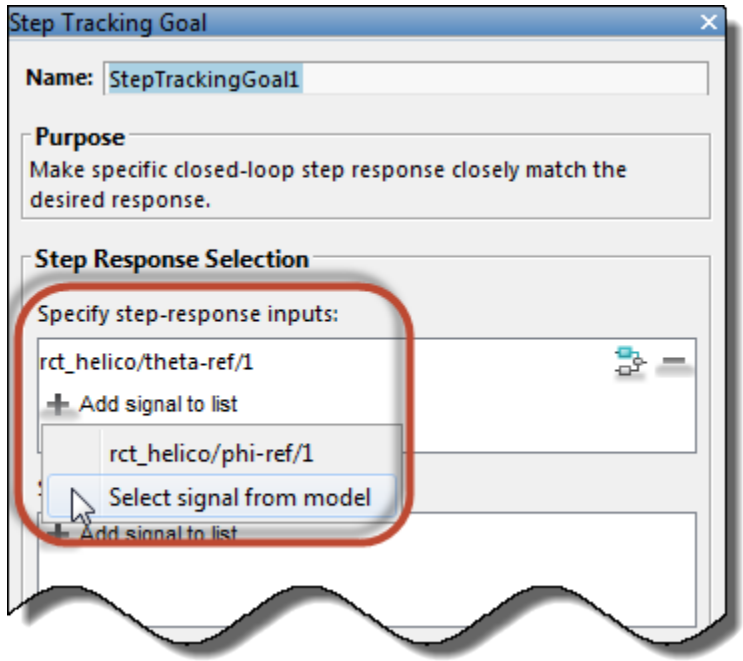
Choose Signal Locations for Evaluating Tuning Goal

Specify the signal locations in your control system at which the tuning goal is evaluated. For example, the step response goal specifies that a step signal applied at a particular input location yields a desired response at a particular output location. Use the **Step Response Selection** section of the dialog box to specify these input and output

locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

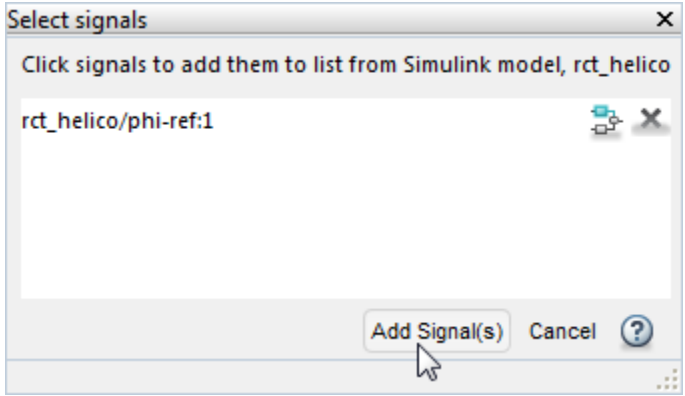
Under **Specify step-response inputs**, click **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.



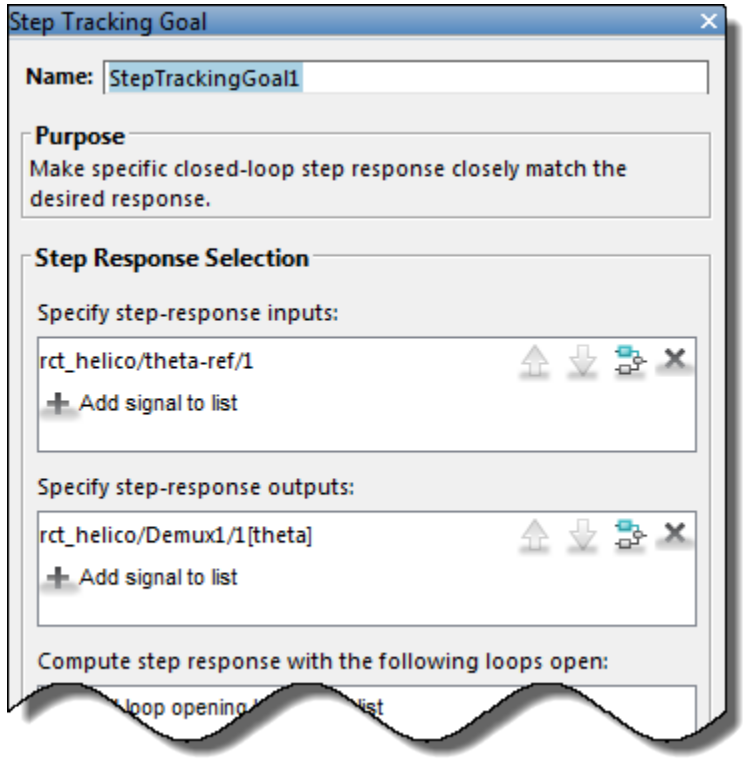
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO tuning goal, and click multiple signals to create a MIMO tuning goal.

Click **Add signal(s)**. The **Select signals** dialog box closes, returning you to the new tuning-goal specification dialog box.



The signals you selected now appear in the list of step-response inputs in the tuning goal dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration constrains the response to a step input applied at `theta-ref` and measured at `theta` in the Simulink model `rct_helico`.



Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and .

Specify Loop Openings

Most tuning goals can be enforced with loops open at one or more locations in the control system. Click **Add loop opening location to list** to specify such locations for the tuning goal.

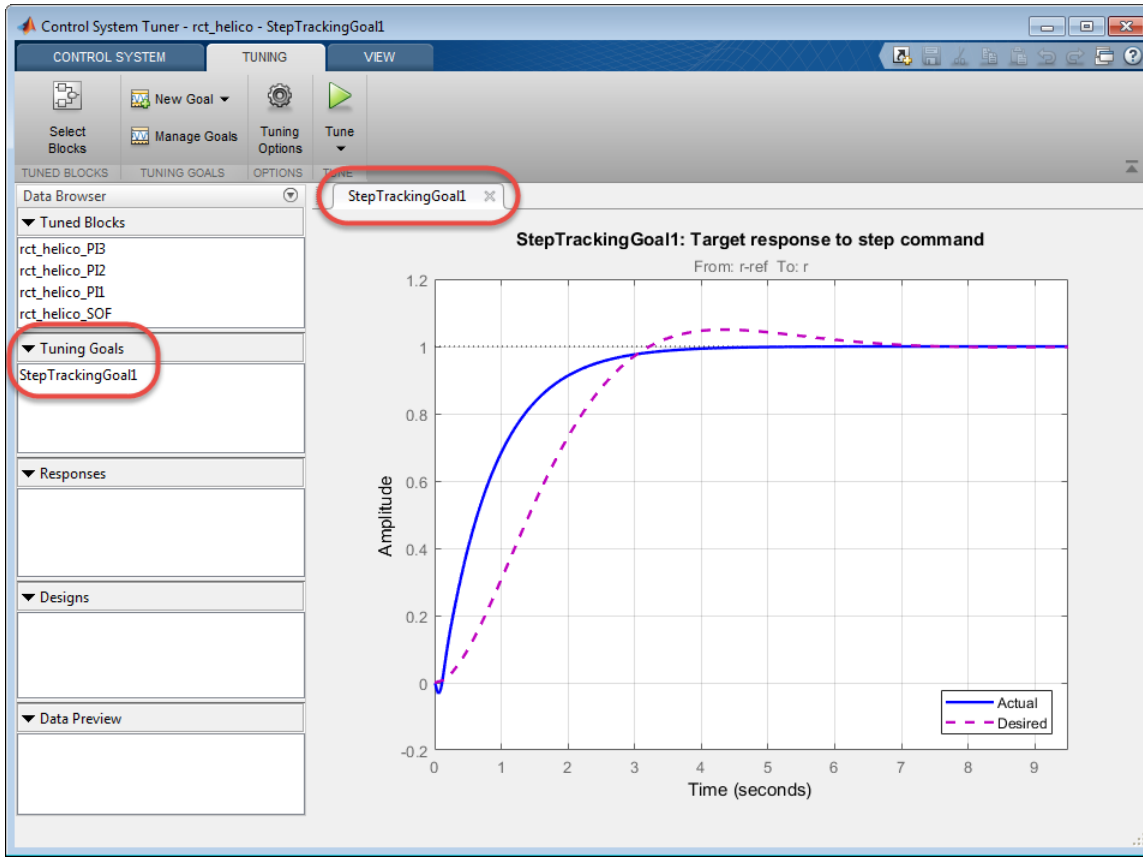
Define Other Specifications of the Tuning Goal

The tuning goal dialog box prompts you to specify other details about the tuning goal. For example, to create a step response requirement, you provide details of the desired step response in the **Desired Response** area of the **Step Response Goal** dialog box. Some tuning goals have additional options in an **Options** section of the dialog box.

For information about the fields for specifying a particular tuning goal, click  in the tuning goal dialog box.

Store the Tuning Goal for Tuning

When you have finished specifying the tuning goal, click **OK** in the tuning goal dialog box. The new tuning goal appears in the **Tuning Goals** section of the Data Browser. A new figure opens displaying a graphical representation of the tuning goal. When you tune your control system, you can refer to this figure to evaluate graphically how closely the tuned system satisfies the tuning goal.



Tip To edit the specifications of the tuning goal, double-click the tuning goal in the Data Browser.

Activate the Tuning Goal for Tuning

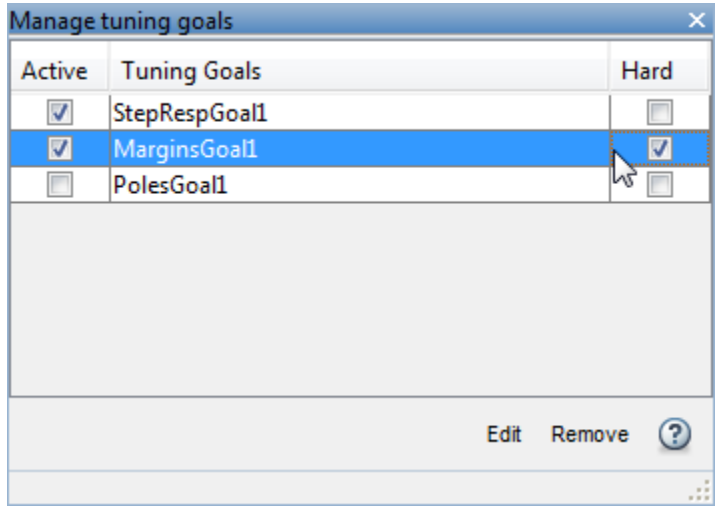
When you have saved your tuning goal, click **New Goal** to create additional tuning goals.

When you are ready to tune your control system, click **Manage Goals** to select which tuning goals are active for tuning. In the **Manage Tuning Goals** dialog box,

Active is checked by default for any new goals. Uncheck **Active** for any tuning goal that you do not want enforced.

You can also designate one or more tuning goals as **Hard** goals. Control System Tuner attempts to satisfy hard requirements, and comes as close as possible to satisfying remaining (soft) requirements subject to the hard constraints. By default, new goals are designated soft goals. Check **Hard** for any goal to designate it a hard goal.

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.



Deactivating tuning goals or designating some goals as soft requirements can be useful when investigating the tradeoffs between different tuning requirements. For example, if you do not obtain satisfactory performance with all your tuning goals active and hard, you might try another design in which less crucial goals are designated as soft or deactivated entirely.

See Also

Related Examples

- “Manage Tuning Goals” on page 9-177

- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 9-49
- “Create Response Plots in Control System Tuner” on page 9-197

Quick Loop Tuning of Feedback Loops in Control System Tuner

This example shows how to tune a Simulink model of a control system to meet a specified bandwidth and specified stability margins in Control System Tuner, without explicitly creating tuning goals that capture these requirements. You can use a similar approach for quick loop tuning of control systems modeled in MATLAB.

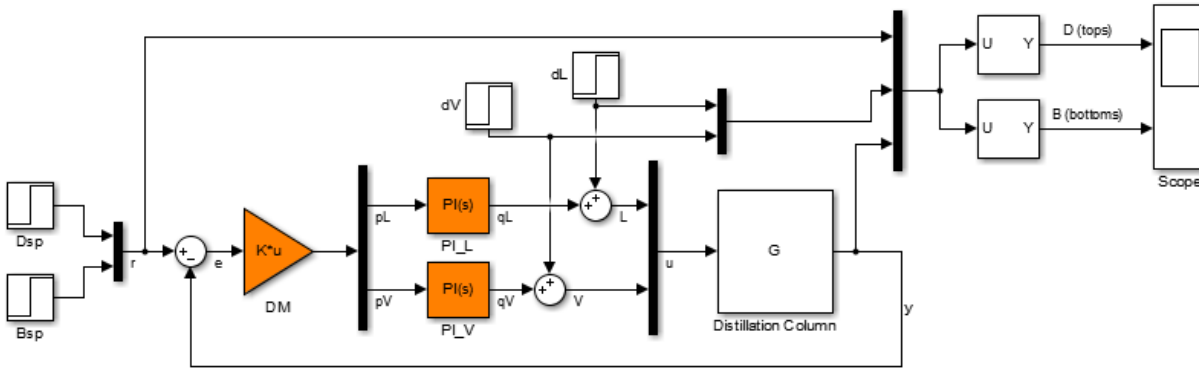
This example demonstrates how the **Quick Loop Tuning** option of Control System Tuner generates tuning goals from your crossover frequency and gain and phase margin specifications. This option lets you quickly set up SISO or MIMO feedback loops for tuning using a loop-shaping approach. The example also shows how to add further tuning requirements to the control system after using the **Quick Loop Tuning** option.

Quick Loop Tuning is the Control System Tuner equivalent of the `looptune` command.

Set up the Model for Tuning

Open the Simulink model.

```
open_system('rct_distillation')
```



This model represents a distillation column, captured in the two-input, two-output plant `G`. The tunable elements are the decoupling gain matrix `DM`, and the two PI controllers, `PI_L` and `PI_V`. (For more information about this model, see “Decoupling Controller for a Distillation Column” (Control System Toolbox).)

Suppose your goal is to tune the MIMO feedback loop between `r` and `y` to a bandwidth between 0.1 and 0.5 rad/s. Suppose you also require a gain margin of 7 dB and a phase margin of 45 degrees. You can use the **Quick Loop Tuning** option to quickly configure Control System Tuner for these goals.

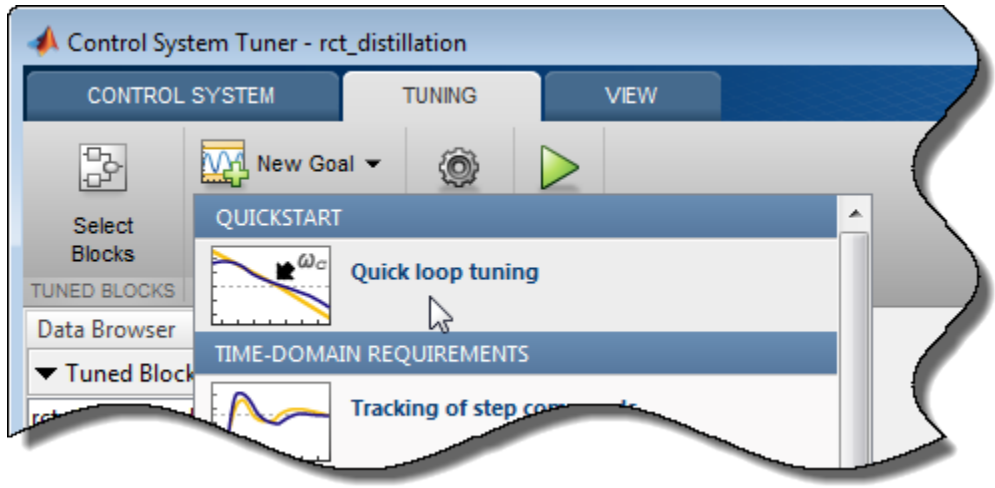
In the Simulink model editor, open Control System Tuner by selecting **Analysis** > **Control Design** > **Control System Tuner**.

Designate the blocks you want to tune. In the **Tuning** tab of Control System Tuner, click **Select Blocks**. In the **Select tuned blocks** dialog box, click **Add blocks**. Then, select `DM`, `PI_L`, and `PI_V` for tuning. (For more information about selecting tuned blocks, see "Specify Blocks to Tune in Control System Tuner" on page 9-25.)

The model is now ready to tune to the target bandwidth and stability margins.

Specify the Goals for Quick Loop Tuning

In the **Tuning** tab, select **New Goal** > **Quick Loop Tuning**.

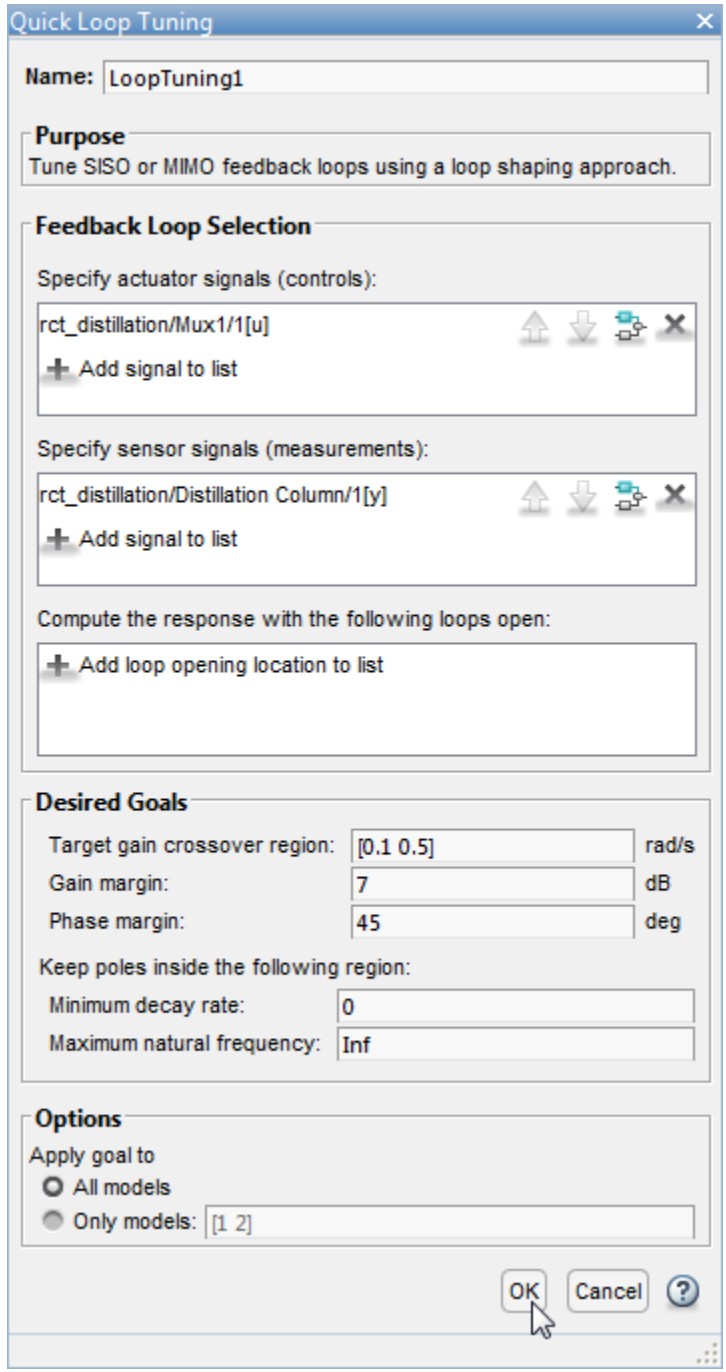


For Quick Loop Tuning, you need to identify the actuator signals and sensor signals that separate the plant portion of the control system from the controller, which for the purpose of Quick Loop Tuning is the rest of the control system. The actuator signals are the controller outputs that drive the plant, or the plant inputs. The sensor signals are the measurements of plant output that feed back into the controller. In this control system,

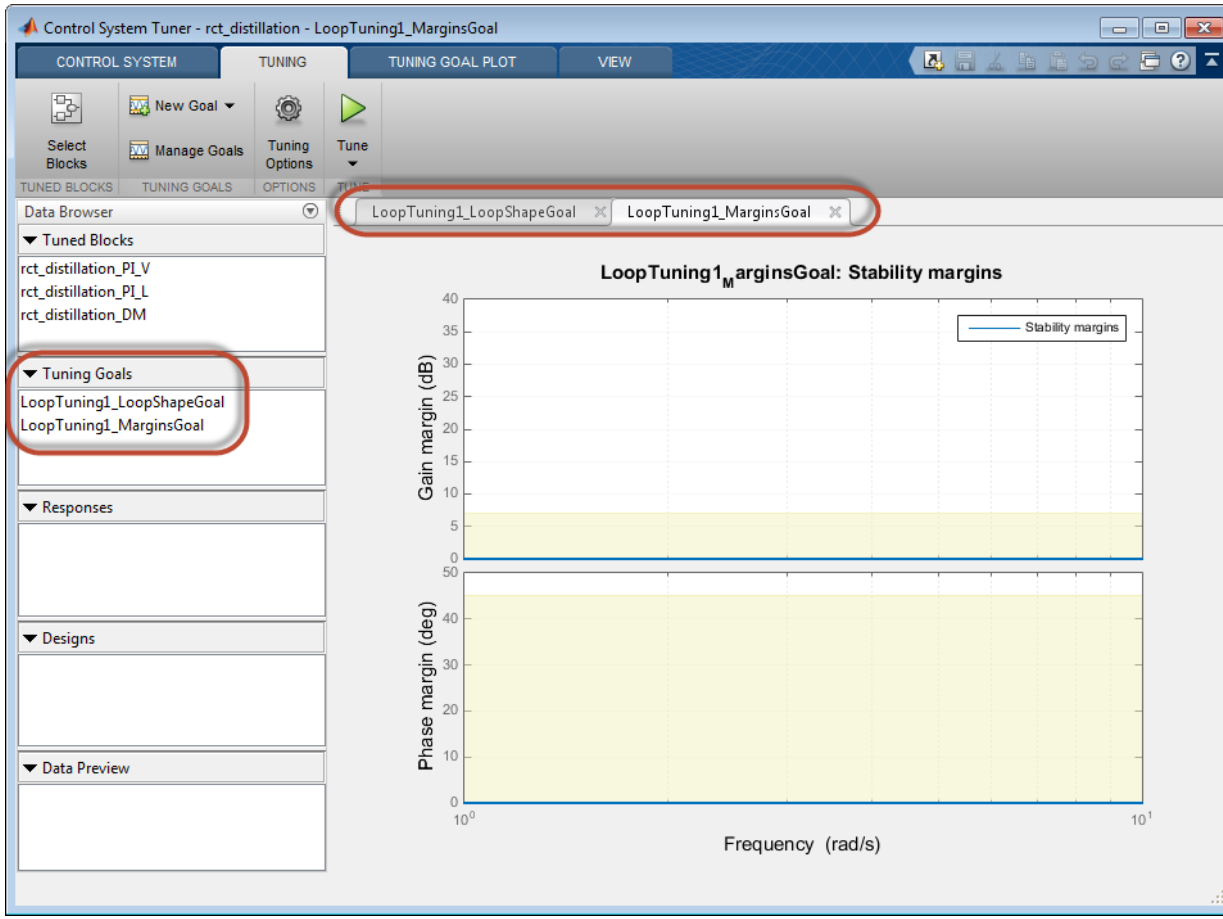
the actuator signals are represented by the vector signal `u`, and the sensor signals by the vector signal `y`.

In the **Quick Loop Tuning** dialog box, under **Specify actuator signals (controls)**, add the actuator signal, `u`. Similarly, under **Specify sensor signals (measurements)**, add the sensor signal, `y` (For more information about specifying signals for tuning, see "Specify Goals for Interactive Tuning" on page 9-40.)

Under **Desired Goals**, in the **Target gain crossover region** field, enter the target bandwidth range, `[0.1 0.5]`. Enter the desired gain margin and phase margin in the corresponding fields.


Quick Loop Tuning
Name: LoopTuning1
Purpose
Tune SISO or MIMO feedback loops using a loop shaping approach.
Feedback Loop Selection
Specify actuator signals (controls):
rct_distillation/Mux1/1[u]
Add signal to list
Specify sensor signals (measurements):
rct_distillation/Distillation Column/1[y]
Add signal to list
Compute the response with the following loops open:
Add loop opening location to list
Desired Goals
Target gain crossover region: [0.1 0.5] rad/s
Gain margin: 7 dB
Phase margin: 45 deg
Keep poles inside the following region:
Minimum decay rate: 0
Maximum natural frequency: Inf
Options
Apply goal to
All models
Only models: [1 2]
OK
Cancel

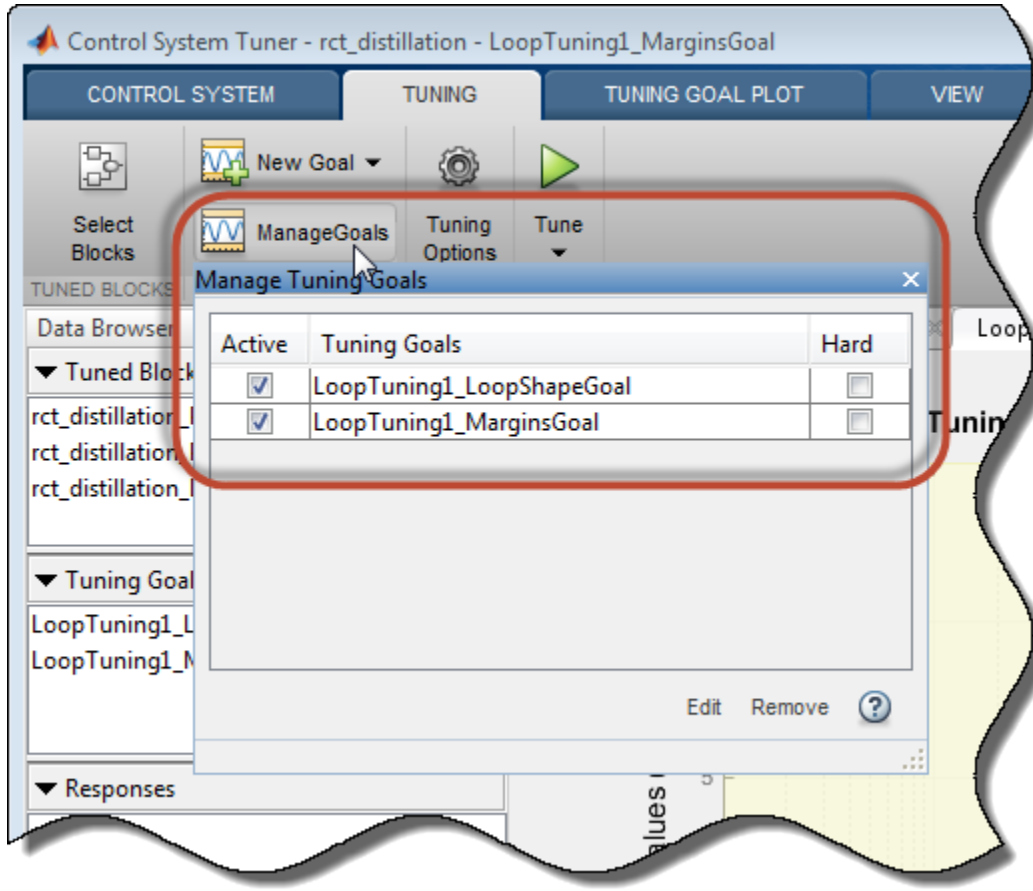
Click **OK**. Control System Tuner automatically generates tuning goals that capture the desired goals you entered in the dialog box.



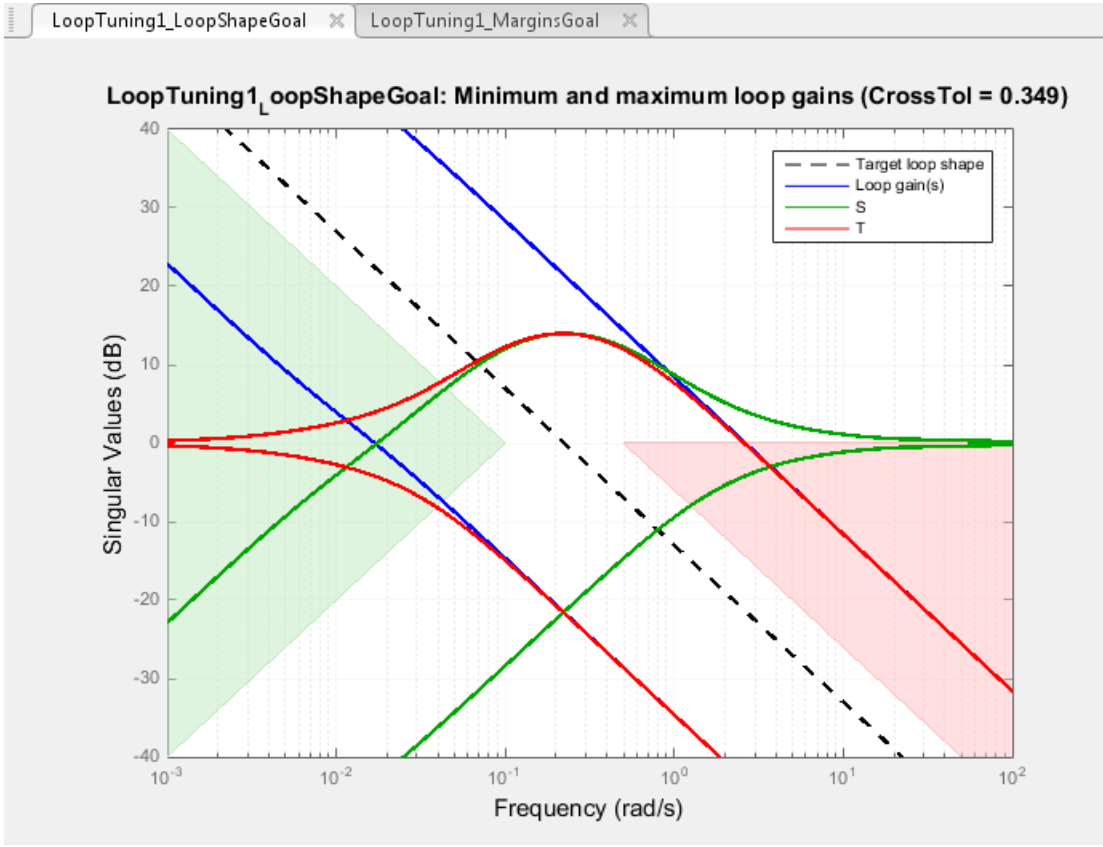
Examine the Automatically-Created Tuning Goals

In this example, Control System Tuner creates a Loop Shape Goal and a Margins Goal. If you had changed the pole-location settings in the **Quick Loop Tuning** dialog box, a Poles goal would also have been created.

Click  **Manage Goals** to examine the automatically-created goals. By default, the goals are active and designated as soft tuning goals.



You can double-click the tuning goals to examine their parameters, which are automatically computed and populated. You can also examine the graphical representations of the tuning goals. In the **Tuning** tab, examine the **LoopTuning1_LoopShapeGoal** plot.



The target crossover range is expressed as a Loop Shape goal with an integrator open-loop gain profile. The shaded areas of the graph show that the permitted crossover range is `[0.1 0.5]` rad/s, as you specified in the **Quick Loop Tuning** dialog box.

Similarly, your margin requirements are captured in the **LoopTuning1_MarginsGoal** plot.

Tune the Model

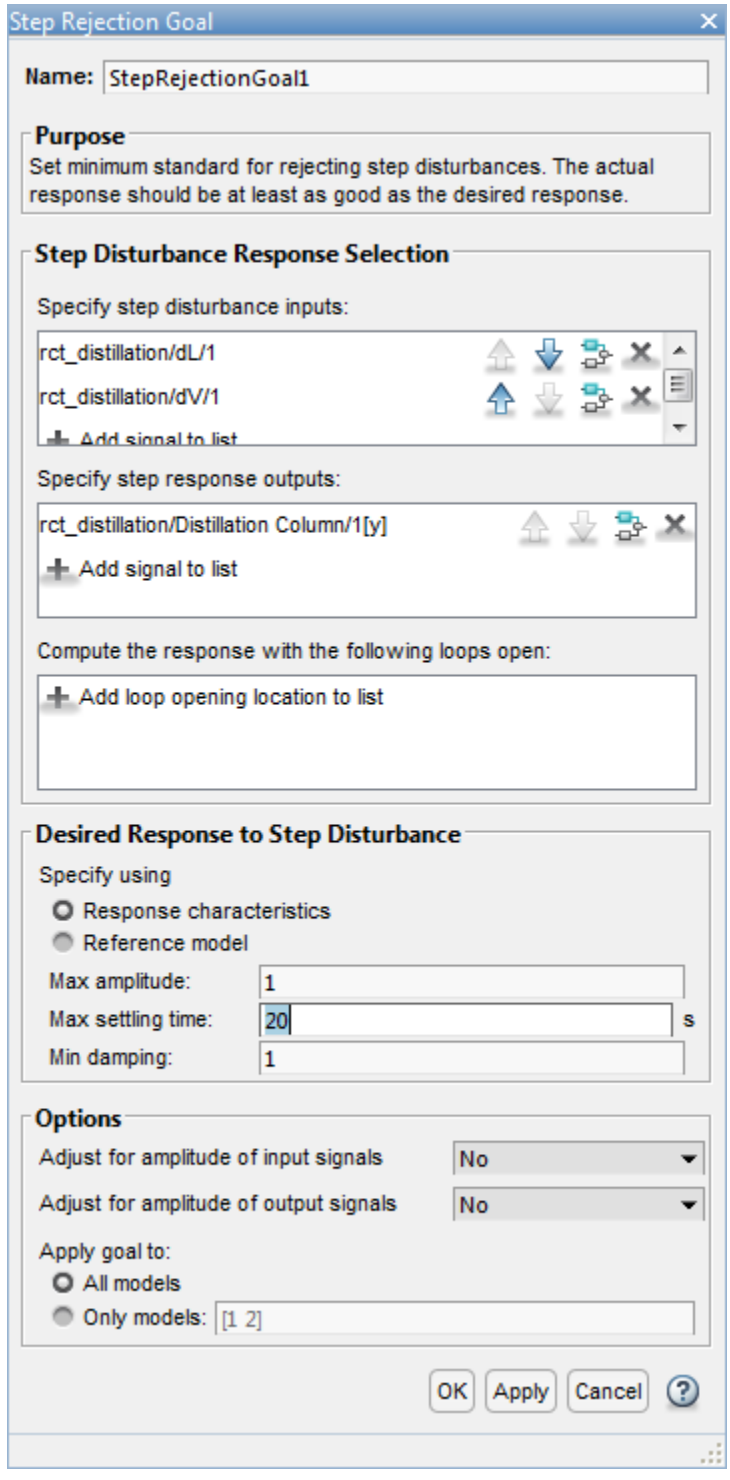
Click  **Tune** to tune the model to meet the automatically-created tuning goals. In the tuning goal plots, you can see that the requirements are satisfied.

To create additional plots for examining other system responses, see “Create Response Plots in Control System Tuner” on page 9-197.

Change Design Requirements

If you want to change your design requirements after using Quick Loop Tuning, you can edit the automatically-created tuning goals and tune the model again. You can also create additional tuning goals.

For example, add a requirement that limits the response to a disturbance applied at the plant inputs. Limit the response to a step command at `dL` and `dV` at the outputs, `y`, to be well damped, to settle in less than 20 seconds, and not exceed 4 in amplitude. Select **New Goal > Rejection of step disturbances** and enter appropriate values in the Step Rejection Goal dialog box. (For more information about creating tuning goals, see “Specify Goals for Interactive Tuning” on page 9-40.)


Step Rejection Goal
Name: StepRejectionGoal1
Purpose
Set minimum standard for rejecting step disturbances. The actual response should be at least as good as the desired response.
Step Disturbance Response Selection
Specify step disturbance inputs:
rct_distillation/dL/1
rct_distillation/dV/1
Add signal to list
Specify step response outputs:
rct_distillation/Distillation Column/1[y]
Add signal to list
Compute the response with the following loops open:
Add loop opening location to list
Desired Response to Step Disturbance
Specify using
Response characteristics
Reference model
Max amplitude: 1
Max settling time: 20 s
Min damping: 1
Options
Adjust for amplitude of input signals No
Adjust for amplitude of output signals No
Apply goal to:
All models
Only models: [1 2]
OK
Apply
Cancel

You can now retune the model to meet all these tuning goals.

See Also

`looptune (for slTuner)`

Related Examples

- “Specify Operating Points for Tuning in Control System Tuner” on page 9-16
- “Manage Tuning Goals” on page 9-177
- “Setup for Tuning Control System Modeled in MATLAB” on page 9-36

Quick Loop Tuning

Purpose

Tune SISO or MIMO feedback loops using a loop-shaping approach in Control System Tuner.

Description

Quick Loop Tuning lets you tune your system to meet open-loop gain crossover and stability margin requirements without explicitly creating tuning goals that capture these requirements. You specify the feedback loop whose open-loop gain you want to shape by designating the actuator signals (controls) and sensor signals (measurements) that form the loop. Actuator signals are the signals that drive the plant. The sensor signals are the plant outputs that feed back into the controller.

You enter the target loop bandwidth and desired gain and phase margins. You can also specify constraints on pole locations of the tuned system, to eliminate fast dynamics. Control System Tuner automatically creates Tuning Goals that capture your specifications and ensure integral action at frequencies below the target loop bandwidth.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Quick Loop Tuning** to specify loop-shaping requirements.

Command-Line Equivalent

When tuning control systems at the command line, use `looptune (for slTuner)` or `looptune` for tuning feedback loops using a loop-shaping approach.

Feedback Loop Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify actuator signals (controls)**

Designate one or more signals in your model as actuator signals. These are the input signals that drive the plant. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Specify sensor signals (measurements)**

  Designate one or more signals in your model as sensor signals. These are the plant outputs that provide feedback into the controller. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Compute the response with the following loops open**

  Designate additional locations at which to open feedback loops for the purpose of tuning the loop defined by the control and measurement signals.

  Quick Loop Tuning tunes the open-loop response of the loop defined by the control and measurement signals. If you want your specifications for that loop to apply with other feedback loops in the system opened, specify loop-opening locations in this section of the dialog box. For example, if you are tuning a cascaded-loop control system with an inner loop and an outer loop, you might want to tune the inner loop with the outer loop open.

For an example showing in more detail how to specify signal locations, see “Specify Goals for Interactive Tuning” on page 9-40.

Desired Goals

Use this section of the dialog box to specify desired characteristics of the tuned system. Control System Tuner converts these into Loop Shape, Margin, and Poles goals.

- **Target gain crossover region**

  Specify a frequency range in which the open-loop gain should cross 0 dB. Specify the frequency range as a row vector of the form `[min,max]`, expressed in frequency units of your model. Alternatively, if you specify a single target frequency, `wc`, the target range is taken as `[wc/10^0.1,wc*10^0.1]`, or `wc` ± 0.1 decade.

- **Gain margin (db)**

  Specify the desired gain margin in decibels. For MIMO control system, the gain margin is the multiloop disk margin. See `loopmargin` for information about multiloop disk margins.

- **Phase margin (degrees)**

  Specify the desired phase margin in degrees. For MIMO control system, the phase margin is the multiloop disk margin. See `loopmargin` for information about multiloop disk margins.

- **Keep poles inside the following region**

  Specify minimum decay rate and maximum natural frequency for the closed-loop poles of the tuned system. While the other Quick Loop Tuning options specify characteristics of the open-loop response, these specifications apply to the closed-loop dynamics.

  The minimum decay rate you enter constrains the closed-loop pole locations to:

  - `Re(s) < -mindecay`, for continuous-time systems.
  - `log(|z|) < -mindecay*Ts`, for discrete-time systems with sample time `Ts`.

  The maximum frequency you enter constrains the closed-loop poles to satisfy `|s| < maxfreq` for continuous time, or `|log(z)| < maxfreq*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the closed-loop system.

Options

Use this section of the dialog box to specify additional characteristics.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Control System Tuner uses `looptuneSetup (for slTuner)` or `looptuneSetup` to convert Quick Loop Tuning specifications into tuning goals.

See Also

Related Examples

- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 9-49
- “Specify Goals for Interactive Tuning” on page 9-40
- “Visualize Tuning Goals” on page 9-188
- “Manage Tuning Goals” on page 9-177

Step Tracking Goal

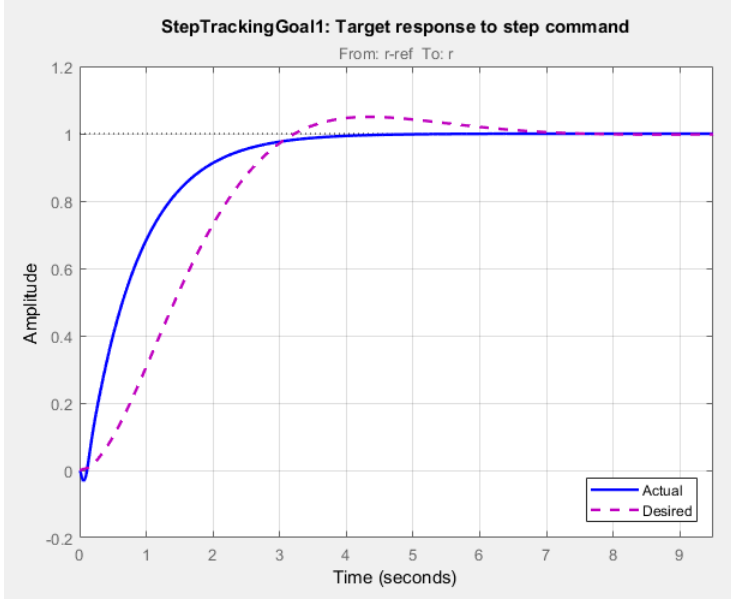
Purpose

Make the step response from specified inputs to specified outputs closely match a target response, when using Control System Tuner.

Description

Step Tracking Goal constrains the step response between the specified signal locations to match the step response of a stable reference system. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify. You can use this goal to constrain a SISO or MIMO response of your control system.

You can specify the reference system for the target step response in terms of first-order system characteristics (time constant) or second-order system characteristics (natural frequency and percent overshoot). Alternatively, you can specify a custom reference system as a numeric LTI model.



Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Tracking of step commands** to create a Step Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepTracking` to specify a step response goal.

Step Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute step response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Desired Response

Use this section of the dialog box to specify the shape of the desired step response.

- **First-order characteristics**

  Specify the desired step response (the reference model H_{ref}) as a first-order response with time constant τ:

  $$H_{ref} = \frac{1/\tau}{s+1/\tau}.$$

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

- **Second-order characteristics**

  Specify the desired step response as a second-order response with time constant τ, and natural frequency $1/\tau$.

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

  Enter the target overshoot percentage in the **Overshoot** text box.

  The second-order reference system has the form:

  $$H_{ref} = \frac{(1/\tau)^2}{s^2 + 2(\zeta/\tau)s + (1/\tau)^2}.$$

  The damping constant ζ is related to the overshoot percentage by $\zeta =$ `cos(atan2(pi,-log(overshoot/100)))`.

- **Custom reference model**

  Specify the reference system for the desired step response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model.

  Enter the name of the reference model in the MATLAB workspace in the **LTI model to match** text field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`.

The reference model must be stable and must have DC gain of 1 (zero steady-state error). The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z = 0`.

The reference model can be MIMO, provided that it is square and that its DC singular value (`sigma`) is 1. Then number of inputs and outputs of the reference model must match the dimensions of the inputs and outputs specified for the step response goal.

For best results, the reference model should also include intrinsic system characteristics such as non-minimum-phase zeros (undershoot).

If your selected inputs and outputs define a MIMO system and you apply a SISO reference system, the software attempts to match the diagonal channels of the MIMO system. In that case, cross-couplings tend to be minimized.

Options

Use this section of the dialog box to specify additional characteristics of the step response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) step response and the target step response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$e_{rel} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| 1 - y_{ref}(t) \right\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref}(t)$ is the step-tracking error of the target model. $\| \cdot \|_2$ denotes the signal energy (2-norm).

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Step Response Goal**, $f(x)$ is given by:

$$f(x)=\frac{\left\|\frac{1}{s}\left(T(s,x)-H_{ref}(s)\right)\right\|_2}{e_{rel}\left\|\frac{1}{s}\left(H_{ref}(s)-I\right)\right\|_2}.$$

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $H_{ref}(s)$ is the reference model. e_{rel} is the relative error

(see "Options" on page 9-66). $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40
- "Manage Tuning Goals" on page 9-177
- "Visualize Tuning Goals" on page 9-188

Step Rejection Goal

Purpose

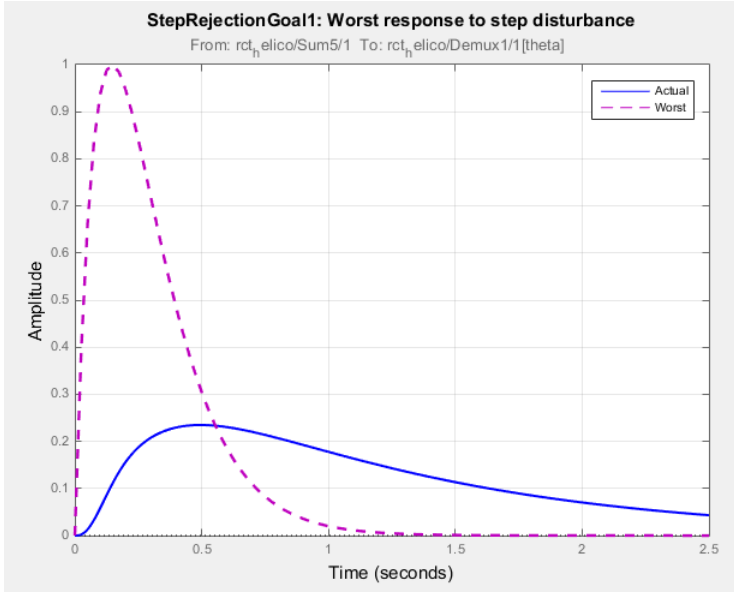
Set a minimum standard for rejecting step disturbances, when using Control System Tuner.

Description

Use **Step Rejection Goal** to specify how a step disturbance injected at a specified location in your control system affects the signal at a specified output location.

You can specify the desired response in time-domain terms of peak value, settling time, and damping ratio. Control System Tuner attempts to make the actual rejection at least as good as the desired response. Alternatively, you can specify the response as a stable reference model having DC-gain. In that case, the tuning goal is to reject the disturbance as well as or better than the reference model.

To specify disturbance rejection in terms of a frequency-domain attenuation profile, use **Disturbance Rejection Goal**.



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The dotted line shows the target step response you specify. The solid line is the current corresponding response of your system.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Rejection of step disturbance** to create a Step Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepRejection` to specify a step response goal.

Step Disturbance Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step disturbance inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step-disturbance response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step response outputs**

  Select one or more signal locations in your model at which to measure the response to the step disturbance. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you

identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Desired Response to Step Disturbance

Use this section of the dialog box to specify the shape of the desired response to the step disturbance. Control System Tuner attempts to make the actual response at least as good as the desired response.

- **Response Characteristics**

  Specify the desired response in terms of time-domain characteristics. Enter the maximum amplitude, maximum settling time, and minimum damping constant in the text boxes.

- **Reference Model**

  Specify the desired response in terms of a reference model.

  Enter the name of the reference model in the MATLAB workspace in the **Reference Model** text field. Alternatively, enter a command to create a suitable reference model, such as `tf([1 0],[1 1.414 1])`.

  The reference model must be stable and must have zero DC gain. The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z` = 0.

  For best results, the reference model and the open-loop response from the disturbance to the output should have similar gains at the frequency where the reference model gain peaks.

Options

Use this section of the dialog box to specify additional characteristics of the step rejection goal.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

When these options are set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitudes of input signals** text box.

Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

The default value, `No`, means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

Step Rejection Goal aims to keep the gain from disturbance to output below the gain of the reference model. The scalar value of the requirement $f(x)$ is given by:

$$f(x) = \left\| W_F(s)\, T_{dy}(s,x) \right\|_\infty ,$$

or its discrete-time equivalent. Here, $T_{dy}(s,x)$ is the closed-loop transfer function of the constrained response, and $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`). W_F is a frequency weighting function derived from the step-rejection profile you specify in the tuning goal. The gain of W_F roughly matches the inverse of the reference model for gain values within 60 dB of the peak gain. For numerical reasons, the weighting function levels off outside this range, unless you specify a reference model that changes slope outside this range. This adjustment is called regularization. Because poles of W_F close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify reference models with very low-frequency or very high-frequency dynamics.For more information about regularization and its effects, see “Visualize Tuning Goals” on page 9-188.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Transient Goal

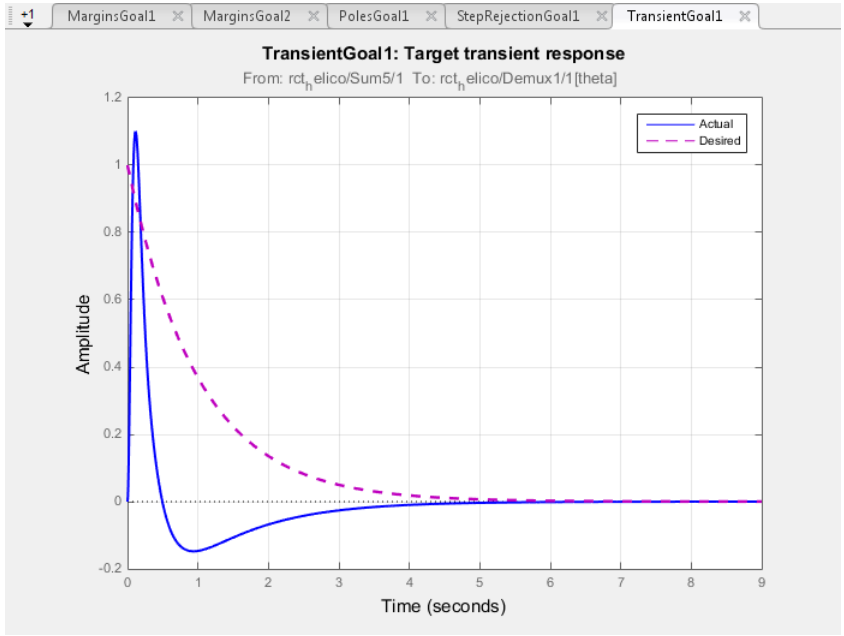
Purpose

Shape how the closed-loop system responds to a specific input signal when using Control System Tuner. Use a reference model to specify the desired transient response.

Description

Transient Goal constrains the transient response from specified input locations to specified output locations. This requirement specifies that the transient response closely match the response of a reference model. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify.

You can constrain the response to an impulse, step, or ramp input signal. You can also constrain the response to an input signal that is given by the impulse response of an input filter you specify.



Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Transient response matching** to create a Transient Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Transient` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify response inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify response outputs**

  Select one or more signal locations in your model at which to measure the transient response. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Initial Signal Selection

Select the input signal shape for the transient response you want to constrain in Control System Tuner.

- `Impulse` — Constrain the response to a unit impulse.
- `Step` — Constrain the response to a unit step. Using `Step` is equivalent to using a Step Tracking Goal.
- `Ramp` — Constrain the response to a unit ramp, `u = t`.
- `Other` — Constrain the response to a custom input signal. Specify the custom input signal by entering a transfer function (`tf` or `zpk`model) in the **Use impulse response of filter** field. The custom input signal is the response of this transfer function to a unit impulse.

  This transfer function represents the Laplace transform of the desired custom input signal. For example, to constrain the transient response to a unit-amplitude sine wave of frequency `w`, enter `tf(w,[1,0,w^2])`. This transfer function is the Laplace transform of sin(*wt*).

  The transfer function you enter must be continuous, and can have no poles in the open right-half plane. The series connection of this transfer function with the reference system for the desired transient response must have no feedthrough term.

Desired Transient Response

Specify the reference system for the desired transient response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model. The Transient Goal constrains the system response to closely match the response of this system to the input signal you specify in **Initial Signal Selection**.

Enter the name of the reference model in the MATLAB workspace in the **Reference Model** field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`. The reference model must be stable, and the series connection of the reference model with the input shaping filter must have no feedthrough term.

Options

Use this section of the dialog box to specify additional characteristics of the transient response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) transient response and the target response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$\text{gap} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| y_{ref(tr)}(t) \right\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref} (deviation from steady-state value or trajectory). $\|\cdot\|_2$ denotes the signal energy (2-norm). The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  When these options are set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitudes of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

  The default value, `No`, means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 9-80), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

  When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

  To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 9-27.

- This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened

at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

For **Transient Goal**, $f(x)$ is based upon the relative gap between the tuned response and the target response:

$$\text{gap} = \frac{\left\| y(t) - y_{ref}(t) \right\|_2}{\left\| y_{ref(tr)}(t) \right\|_2}.$$

$y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref}

(deviation from steady-state value or trajectory). $\|\cdot\|_2$ denotes the signal energy (2-norm). The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

LQR/LQG Goal

Purpose

Minimize or limit Linear-Quadratic-Gaussian (LQG) cost in response to white-noise inputs, when using Control System Tuner.

Description

LQR/LQG Goal specifies a tuning requirement for quantifying control performance as an LQG cost. It is applicable to any control structure, not just the classical observer structure of optimal LQG control.

The LQG cost is given by:

$J = E(z(t)'\ QZ\ z(t))$.

$z(t)$ is the system response to a white noise input vector $w(t)$. The covariance of $w(t)$ is given by:

$E(w(t)w(t)') = QW$.

The vector $w(t)$ typically consists of external inputs to the system such as noise, disturbances, or command. The vector $z(t)$ includes all the system variables that characterize performance, such as control signals, system states, and outputs. $E(x)$ denotes the expected value of the stochastic variable x.

The cost function J can also be written as an average over time:

$$J = \lim_{T\to\infty} E\left(\frac{1}{T}\int_0^T z(t)'\ QZ\ z(t)\,dt\right).$$

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > LQR/LQG objective** to create an LQR/LQG Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LQG` to specify an LQR/LQG goal.

Signal Selection

Use this section of the dialog box to specify noise input locations and performance output locations. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify noise inputs (w)**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the LQG cost for a noise input `'u'` and performance output `'y'`, click **Add signal to list** and select `'u'`. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Specify performance outputs (z)**

  Select one or more signal locations in your model as performance outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the LQG cost for a noise input `'u'` and performance output `'y'`, click **Add signal to list** and select `'y'`. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate LQG objective with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

LQG Objective

Use this section of the dialog box to specify the noise covariance and performance weights for the LQG goal.

- **Performance weight Qz**

  Performance weights, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix. Use a diagonal matrix to independently scale or penalize the contribution of each variable in *z*.

The performance weights contribute to the cost function according to:

$J = E(z(t)'\ Qz\ z(t))$.

When you use the LQG goal as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select Qz values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

- **Noise Covariance Qw**

  Covariance of the white noise input vector $w(t)$, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix with as many rows as there are entries in the vector $w(t)$. A diagonal matrix means the entries of $w(t)$ are uncorrelated.

  The covariance of $w(t$ is given by:

  $E(w(t)w(t)') = QW$.

  When you are tuning a control system in discrete time, the LQG goal assumes:

  $E(w[k]w[k]') = QW/T_s$.

  T_s is the model sample time. This assumption ensures consistent results with tuning in the continuous-time domain. In this assumption, $w[k]$ is discrete-time noise obtained by sampling continuous white noise $w(t)$ with covariance QW. If in your system $w[k]$ is a truly discrete process with known covariance QWd, use the value T_s*QWd for the QW value.

Options

Use this section of the dialog box to specify additional characteristics of the LQG goal.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For

example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal, is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 9-27.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **LQR/LQG Goal**, $f(x)$ is given by the cost function J:

$J = E(z(t)' \ Qz \ z(t))$.

When you use the LQG requirement as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select `Qz` values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177

Gain Goal

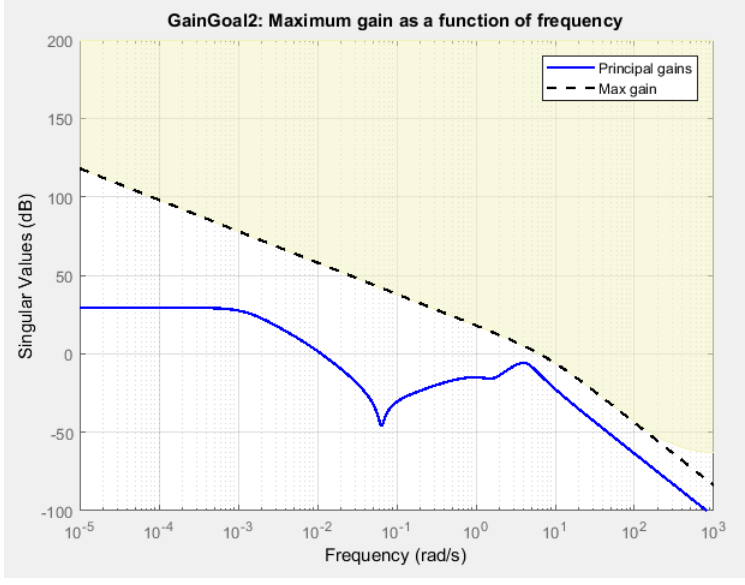
Purpose

Limit gain of a specified input/output transfer function, when using Control System Tuner.

Description

Gain Goal limits the gain from specified inputs to specified outputs. If you specify multiple inputs and outputs, Gain Goal limits the largest singular value of the transfer matrix. (See `sigma` for more information about singular values.) You can specify a constant maximum gain at all frequencies. Alternatively, you can specify a frequency-dependent gain profile.

Use Gain Goal, for example, to enforce a custom roll-off rate in a particular frequency band. To do so, specify a maximum gain profile in that band. You can also use Gain Goal to enforce disturbance rejection across a particular input/output pair by constraining the gain to be less than 1.



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the gain goal is not satisfied.

By default, Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Gain limits** to create a Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Gain` to specify a maximum gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Options

Use this section of the dialog box to specify additional characteristics of the gain goal.

- **Limit gain to**

  Enter the maximum gain in the text box. You can specify a scalar value or a frequency-dependent gain profile. To specify a frequency-dependent gain profile, enter a SISO numeric LTI model. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify a gain profile that rolls off at –40dB/decade in the frequency band from 8 to 800 rad/s, enter `frd([0.8 8 800],[10 1 1e-4])`.

  You must specify a SISO transfer function. If you specify multiple input signals or output signals, the gain profile applies to all I/O pairs between these signals.

  If you are tuning in discrete time, you can specify the maximum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the gain profile in discrete time gives you more control over the gain profile near the Nyquist frequency.

- **Stabilize I/O transfer**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Adjust for signal amplitude**

  When this option is set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitude of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_F(s) D_o^{-1} T(s,x) D_i \right\|_\infty ,$$

or its discrete-time equivalent. Here, $T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. D_o and D_i are the scaling matrices described in "Options" on page 9-88. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `getPeakGain`).

The frequency weighting function W_F is the regularized gain profile, derived from the maximum gain profile you specify. The gain of W_F roughly matches the inverse of the gain profile you specify, inside the frequency band you set in the **Enforce goal in frequency range** field of the tuning goal. W_F is always stable and proper. Because poles of $W_F(s)$ close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify maximum gain profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see "Visualize Tuning Goals" on page 9-188.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Variance Goal

Purpose

Limit white-noise impact on specified output signals, when using Control System Tuner.

Description

Variance Goal imposes a noise attenuation constraint that limits the impact on specified output signals of white noise applied at specified inputs. The noise attenuation is measured by the ratio of the noise variance to the output variance.

For stochastic inputs with a nonuniform spectrum (colored noise), use “Weighted Variance Goal” on page 9-123 instead.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Signal variance attenuation** to create a Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Variance` to specify a constraint on noise amplification.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.
- **Specify stochastic outputs**

Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Options

Use this section of the dialog box to specify additional characteristics of the variance goal.

- **Attenuate input variance by a factor**

  Enter the desired noise attenuation from the specified inputs to outputs. This value specifies the maximum ratio of noise variance to output variance.

  When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous, and interprets the desired noise attenuation as a bound on the continuous-time H_2 norm. This ensures that continuous-time and discrete-time tuning give consistent results. If the plant and noise processes are truly discrete, and you want to bound the discrete-time H_2 norm instead, multiple the desired attenuation value by $\sqrt{T_s}$. T_s is the sample time of the model you are tuning.

- **Adjust for signal amplitude**

  When this option is set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitude of input signals** text box.

Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 9-95), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

  When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall

feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 9-27.

- This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Variance Goal**, $f(x)$ is given by:

$$f(x) = \left\| \text{Attenuation} \cdot T(s,x) \right\|_2 .$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \left\| \frac{\text{Attenuation}}{\sqrt{T_s}} T(z,x) \right\|_2 .$$

T_s is the sample time of the discrete-time transfer function $T(z,x)$.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Reference Tracking Goal

Purpose

Make specified outputs track reference inputs with prescribed performance and fidelity, when using Control System Tuner. Limit cross-coupling in MIMO systems.

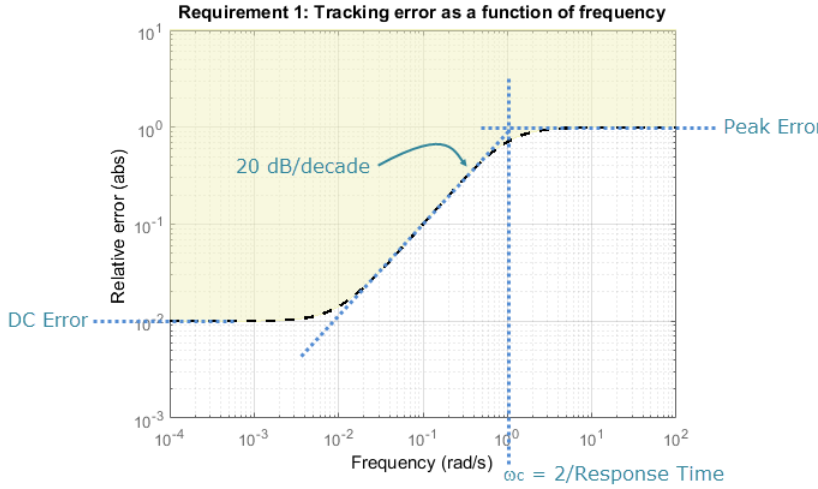
Description

Reference Tracking Goal constrains tracking between the specified signal locations. The constraint is satisfied when the maximum relative tracking error falls below the value you specify at all frequencies. The relative error is the gain from reference input to tracking error as a function of frequency.

You can specify the maximum error profile directly as a function of frequency. Alternatively, you can specify the tracking goal a target DC error, peak error, and response time. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\mathrm{MaxError} = \frac{(\mathrm{PeakError})s + \omega_c(\mathrm{DCError})}{s + \omega_c}.$$

Here, ω_c is 2/(response time). The following plot illustrates these relationships for an example set of values.



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The dotted line shows the error profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Reference Tracking** to create a Reference Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Tracking` to specify a tracking goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify reference inputs**

  Select one or more signal locations in your model as reference signals. To constrain a SISO response, select a single-valued reference signal. For example, to constrain the

response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify reference-tracking outputs**

  Select one or more signal locations in your model as reference-tracking outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Evaluate tracking performance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Tracking Performance

Use this section of the dialog box to specify frequency-domain constraints on the tracking error.

Response time, DC error, and peak error

Select this option to specify the tracking error in terms of response time, percent steady-state error, and peak error across all frequencies. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\mathrm{MaxError} = \frac{(\mathrm{PeakError})s + \omega_c\,(\mathrm{DCError})}{s + \omega_c}.$$

When you select this option, enter the following parameters in the text boxes:

- **Response Time** — Enter the target response time. The tracking bandwidth is given by ω_c = 2/Response Time. Express the target response time in the time units of your model.
- **Steady-state error (%)** — Enter the maximum steady-state fractional tracking error, expressed in percent. For MIMO tracking goals, this steady-state error applies to all I/O pairs. The steady-state error is the DC error expressed as a percentage, `DCError`/100.
- **Peak error across frequency (%)** — Enter the maximum fractional tracking error across all frequencies, expressed in percent.

Maximum error as a function of frequency

Select this option to specify the maximum tracking error profile as a function of frequency.

Enter a SISO numeric LTI model in the text box. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise error profile using an `frd` model. When you do so, the software automatically maps the error profile to a smooth transfer function that approximates the desired error profile. For example, to specify a maximum error of 0.01 below about 1 rad/s, gradually rising to a peak error of 1 at 100 rad/s, enter `frd([0.01 0.01 1],[0 1 100])`.

For MIMO tracking goals, this error profile applies to all I/O pairs.

If you are tuning in discrete time, you can specify the maximum error profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the attenuation profile in continuous time, the tuning software discretizes it. Specifying the error profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the tracking goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and

100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

  The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Tracking Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_F(s)\left(T(s,x) - I\right) \right\|_\infty ,$$

or its discrete-time equivalent. Here, $T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, and $\|\cdot\|_\infty$ denotes the H_∞ norm (see `getPeakGain`). W_F is a frequency weighting function derived from the error profile you specify in the tuning goal. The gain of W_F roughly matches the inverse of the error profile for gain values between –20 dB and 60 dB. For numerical reasons, the weighting function levels off outside this range, unless you specify a reference model that changes slope outside this range. This adjustment is called regularization. Because poles of W_F close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning of the `systune` optimization problem, it is not recommended to specify error profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see "Visualize Tuning Goals" on page 9-188.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Visualize Tuning Goals” on page 9-188
- “Manage Tuning Goals” on page 9-177

Overshoot Goal

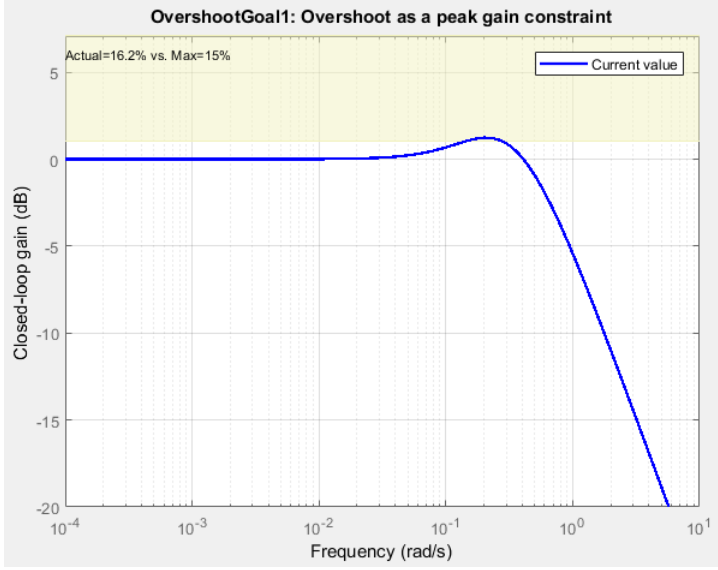
Purpose

Limit overshoot in the step response from specified inputs to specified outputs, when using Control System Tuner.

Description

Overshoot Goal limits the overshoot in the step response between the specified signal locations. The constraint is satisfied when the overshoot in the tuned response is less than the target overshoot

The software maps the maximum overshoot to a peak gain constraint, assuming second-order system characteristics. Therefore, for tuning higher-order systems, the overshoot constraint is only approximate. In addition, the Overshoot Goal cannot reliably reduce the overshoot below 5%.



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Maximum overshoot** to create an Overshoot Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Overshoot` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of outputs.

- **Evaluate overshoot with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Options

Use this section of the dialog box to specify additional characteristics of the overshoot goal.

- **Limit % overshoot to**

  Enter the maximum percent overshoot. Overshoot Goal cannot reliably reduce the overshoot below 5%

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. This tells Control System Tuner to take into account that the first reference signal is 100 times greater than the second reference signal.

  The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Overshoot Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. The percent deviation from $f(x) = 1$ roughly corresponds to the percent deviation from the specified overshoot target. For example, $f(x) = 1.2$ means the actual overshoot exceeds the target by roughly 20%, and $f(x) = 0.8$ means the actual overshoot is about 20% less than the target.

Overshoot Goal uses $\|T\|_\infty$ as a proxy for the overshoot, based on second-order model characteristics. Here, T is the closed-loop transfer function that the requirement

constrains. The overshoot is tuned in the range from 5% ($\|T\|_\infty = 1$) to 100% ($\|T\|_\infty$). **Overshoot Goal** is ineffective at forcing the overshoot below 5%.

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40
- "Manage Tuning Goals" on page 9-177

- “Visualize Tuning Goals” on page 9-188

Disturbance Rejection Goal

Purpose

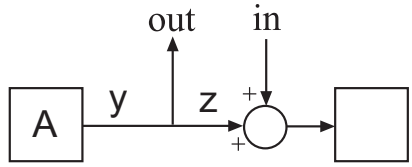
Attenuate disturbances at particular locations and in particular frequency bands, when using Control System Tuner.

Description

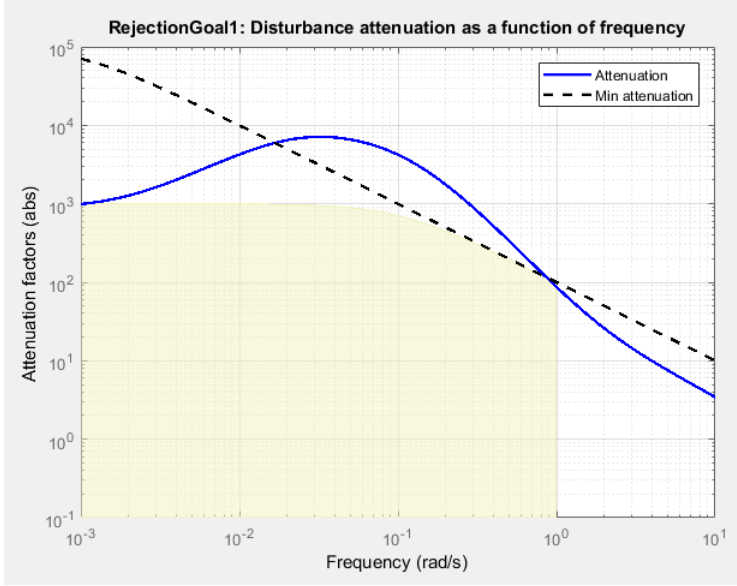
Disturbance Rejection Goal specifies the minimum attenuation of a disturbance injected at a specified location in a control system.

When you use this tuning goal, the software attempts to tune the system so that the attenuation of a disturbance at the specified location exceeds the minimum attenuation factor you specify. This attenuation factor is the ratio between the open- and closed-loop sensitivities to the disturbance, and is a function of frequency.

The following diagram illustrates how the attenuation factor is calculated. Suppose you specify a location in your control system, `y`, which is the output of a block `A`. In that case, the software calculates the closed-loop sensitivity at `out` to a signal injected at `in`. The software also calculates the sensitivity with the control loop opened at the location `z`.



To specify a Disturbance Rejection Goal, you specify one or more locations at which to attenuate disturbance. You also provide the frequency-dependent minimum attenuation factor as a numeric LTI model. You can achieve disturbance attenuation only inside the control bandwidth. The loop gain must be larger than one for the disturbance to be attenuated (attenuation factor > 1).



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied. The solid line is the current corresponding response of your system.

If you prefer to specify sensitivity to disturbance at a location, rather than disturbance attenuation, you can use “Sensitivity Goal” on page 9-114.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Disturbance rejection** to create a Disturbance Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Rejection` to specify a disturbance rejection goal.

Disturbance Scenario

Use this section of the dialog box to specify the signal locations at which to inject the disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Inject disturbances at the following locations**

  Select one or more signal locations in your model at which to measure the disturbance attenuation. To constrain a SISO response, select a single-valued location. For example, to attenuate disturbance at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Rejection Performance

Specify the minimum disturbance attenuation as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired attenuation profile as a function of frequency. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise minimum disturbance rejection using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify an attenuation factor of 100 (40 dB) below 1 rad/s, that gradually drops to 1 (0 dB) past 10 rad/s, enter `frd([100 100 1 1],[0 1 10 100])`.

If you are tuning in discrete time, you can specify the attenuation profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the attenuation profile in continuous time, the tuning software discretizes it. Specifying the attenuation profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the disturbance rejection goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  Regardless of the limits you enter, a disturbance rejection goal can only be enforced within the control bandwidth.

- **Equalize cross-channel effects**

  For multiloop or MIMO disturbance rejection requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Disturbance Rejection Goal**, $f(x)$ is given by:

$$f(x) = \max_{\omega \in \Omega} \left\| W_S(j\omega) S(j\omega, x) \right\|_\infty ,$$

or its discrete-time equivalent. Here, $S(j\omega,x)$ is the closed-loop sensitivity function measured at the disturbance location. Ω is the frequency interval over which the requirement is enforced, specified in the **Enforce goal in frequency range** field. W_S is a frequency weighting function derived from the attenuation profile you specify. The gains of W_S and the specified profile roughly match for gain values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify loop shapes with very low-frequency or very high-frequency dynamics.For more information about regularization and its effects, see “Visualize Tuning Goals” on page 9-188.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Sensitivity Goal

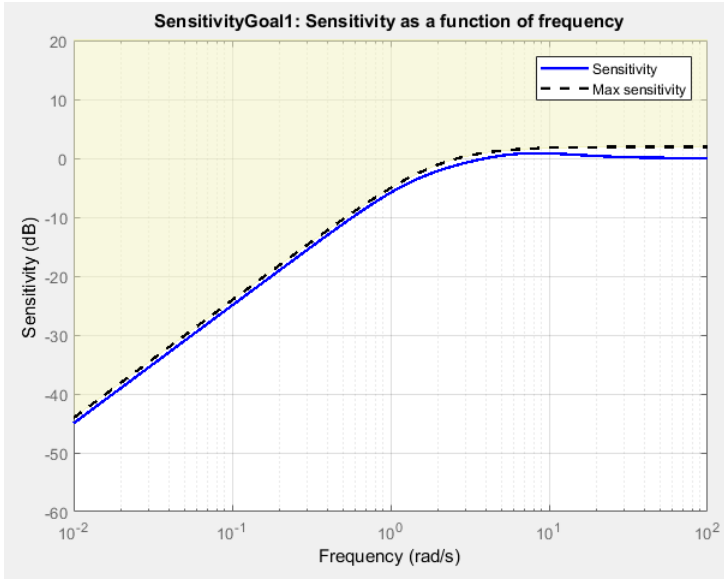
Purpose

Limit sensitivity of feedback loops to disturbances, when using Control System Tuner.

Description

Sensitivity Goal limits the sensitivity of a feedback loop to disturbances. You specify the maximum sensitivity as a function of frequency. Constrain the sensitivity to be smaller than one at frequencies where you need good disturbance rejection.

To specify a Sensitivity Goal, you specify one or more locations at which to limit sensitivity. You also provide the frequency-dependent maximum sensitivity as a numeric LTI model whose magnitude represents the desired sensitivity as a function of frequency.



When you create a tuning goal in Control System Tuner, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

If you prefer to specify disturbance attenuation at a particular location, rather than sensitivity to disturbance, you can use “Disturbance Rejection Goal” on page 9-109.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Sensitivity of feedback loops** to create a Sensitivity Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Sensitivity` to specify a disturbance rejection goal.

Sensitivity Evaluation

Use this section of the dialog box to specify the signal locations at which to compute the sensitivity to disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Measure sensitivity at the following locations**

  Select one or more signal locations in your model at which to measure the sensitivity to disturbance. To constrain a SISO response, select a single-valued location. For example, to limit sensitivity at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Sensitivity Bound

Specify the maximum sensitivity as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired sensitivity bound as a function of frequency. For example, you can specify a smooth transfer

function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum sensitivity using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired sensitivity. For example, to specify a sensitivity that rolls up at 20 dB per decade and levels off at unity above 1 rad/s, enter `frd([0.01 1 1],[0.001 0.1 100])`.

If you are tuning in discrete time, you can specify the maximum sensitivity profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the sensitivity profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the sensitivity goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Equalize cross-channel effects**

  For multiloop or MIMO sensitivity requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Sensitivity Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_S(s)\, S(s,x) \right\|_\infty ,$$

or its discrete-time equivalent. Here, $S(s,x)$ is the closed-loop sensitivity function

measured at the location specified in the tuning goal. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`). W_S is a frequency weighting function derived from the sensitivity profile you specify. The gain of W_S roughly matches the inverse of the specified profile for gain values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify sensitivity profiles with very low-frequency or very high-frequency dynamics.For more information about regularization and its effects, see “Visualize Tuning Goals” on page 9-188.

Implicit Constraint

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Weighted Gain Goal

Purpose

Frequency-weighted gain limit for tuning with Control System Tuner.

Description

Weighted Gain Goal limits the gain of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions that you can use to emphasize particular frequency bands. Weighted Gain Goal constrains the peak gain of $WL(s)H(s)WR(s)$ to values less than 1. If $H(s)$ is a MIMO transfer function, Weighted Gain Goal constrains the largest singular value of $H(s)$.

By default, Weighted Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Frequency-weighted gain limit** to create a Weighted Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedGain` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued

input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

For an example showing in more detail how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 9-40.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. The tuning goal ensures that the gain $H(s)$ from the specified input to output satisfies the inequality:

$$||WL(s)H(s)WR(s)||_\infty < 1.$$

WL provides the weighting for the output channels of $H(s)$, and WR provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify

different weights for each channel by specifying matrices or MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of *WL* and *WR*. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the weighted gain goal.

- **Stabilize I/O transfer**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| WL\ H(s,x)\ WR \right\|_\infty .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. Here, $\left\| \cdot \right\|_\infty$ denotes the H_∞ norm (see `getPeakGain`).

This tuning goal also imposes an implicit stability constraint on the weighted closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40
- "Visualize Tuning Goals" on page 9-188
- "Manage Tuning Goals" on page 9-177

Weighted Variance Goal

Purpose

Frequency-weighted limit on noise impact on specified output signals for tuning with Control System Tuner.

Description

Weighted Variance Goal limits the noise impact on the outputs of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions you can use to model a noise spectrum or emphasize particular frequency bands. Thus, you can use Weighted Variance Goal to tune the system response to stochastic inputs with a nonuniform spectrum such as colored noise or wind gusts.

Weighted Variance minimizes the response to noise at the inputs by minimizing the H_2 norm of the frequency-weighted transfer function. The H_2 norm measures:

- The total energy of the impulse response, for deterministic inputs to the transfer function.
- The square root of the output variance for a unit-variance white-noise input, for stochastic inputs to the transfer function. Equivalently, the H_2 norm measures the root-mean-square of the output for such input.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Frequency-weighted variance attenuation** to create a Weighted Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedVariance` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify stochastic outputs**

  Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal.

WL provides the weighting for the output channels of $H(s)$, and *WR* provides the weighting for the input channels.

You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting as a function of frequency. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s. To limit the response to a nonuniform noise distribution, enter as *WR* an LTI model whose magnitude represents the noise spectrum.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify

different weights for each channel by specifying MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of *WL* and *WR*. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the weighted variance goal.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see “Algorithms” on page 9-126), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. Control System Tuner returns

an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 9-27.

- This tuning goal also imposes an implicit stability constraint on the weighted closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Variance Goal**, $f(x)$ is given by:

$$f(x) = \left\| WL\, H(s,x)\, WR \right\|_2 .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \frac{1}{\sqrt{T_s}} \left\| WL(z)\, H(z,x)\, WR(z) \right\|_2 .$$

T_s is the sample time of the discrete-time transfer function $H(z,x)$.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Visualize Tuning Goals” on page 9-188
- “Manage Tuning Goals” on page 9-177

Minimum Loop Gain Goal

Purpose

Boost gain of feedback loops at low frequency when using Control System Tuner.

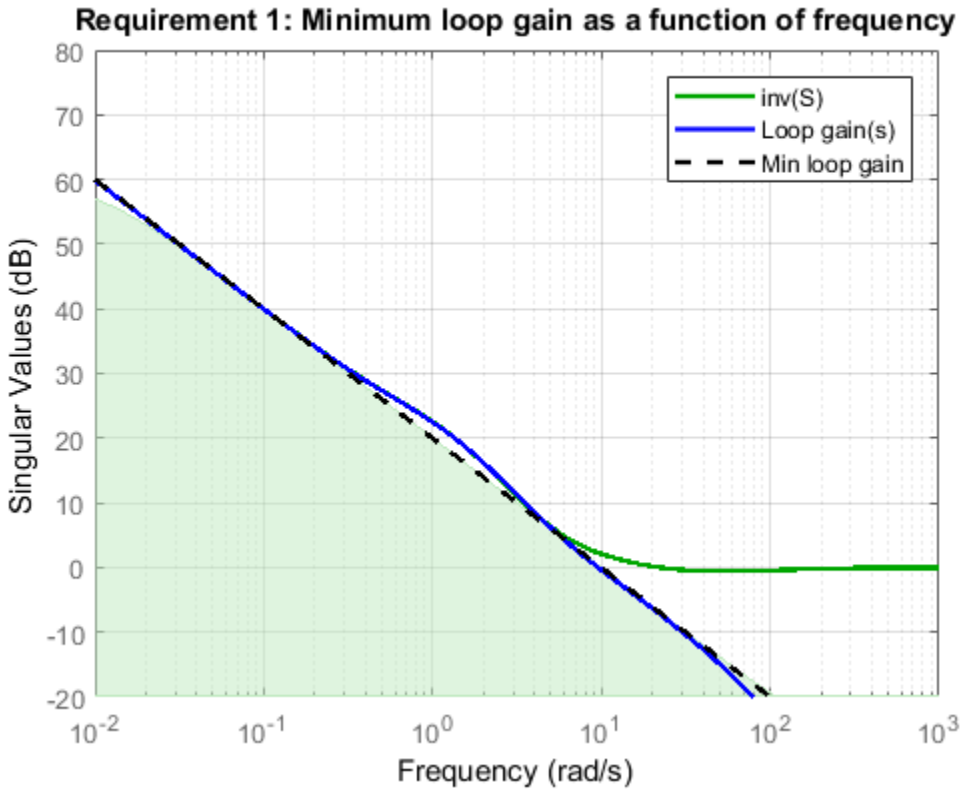
Description

Minimum Loop Gain Goal enforces a minimum loop gain in a particular frequency band. This tuning goal is useful, for example, for improving disturbance rejection at a particular location.

Minimum Loop Gain Goal imposes a minimum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the minimum open-loop gain as a function of frequency (a minimum gain profile). For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of L.

When you tune a control system, the minimum gain profile is converted to a minimum gain constraint on the inverse of the sensitivity function, $\text{inv}(S) = (I + L)$.

The following figure shows a typical specified minimum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The green region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much larger than 1, imposing a minimum gain on $\text{inv}(S)$ is a good proxy for a minimum open-loop gain.



Minimum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Minimum Loop Gain Goal and Maximum Loop Gain Goal specify only low-gain or high-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use “Loop Shape Goal” on page 9-140 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Minimum gain for open-loop response** to create a Minimum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MinLoopGain` to specify a minimum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Desired Loop Gain

Use this section of the dialog box to specify the target minimum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target minimum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a target minimum gain and frequency. For example, to specify an integral gain profile

with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain above** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the minimum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the minimum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum loop gain. For example, to specify minimum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the minimum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the minimum loop gain goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Minimum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_S \left(D^{-1} S D \right) \right\|_\infty .$$

D is a diagonal scaling (for MIMO loops). S is the sensitivity function at `Location`. W_S is a frequency-weighting function derived from the minimum loop gain profile you specify. The gain of this function roughly matches the specified loop gain for values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify gain profiles with very

low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 9-188.

Although S is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing a lower bound on the open-loop transfer function, L, in a frequency band where the gain of L is greater than 1. To see why, note that $S = 1/(1 + L)$. For SISO loops, when $|L| >> 1$, $|S| \approx 1/|L|$. Therefore, enforcing the open-loop minimum gain requirement, $|L| > |W_S|$, is roughly equivalent to enforcing $|W_s S| < 1$. For MIMO loops, similar reasoning applies, with $||S|| \approx 1/\sigma_{\min}(L)$, where $\sigma_{\min}$ is the smallest singular value.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

Maximum Loop Gain Goal

Purpose

Suppress gain of feedback loops at high frequency when using Control System Tuner.

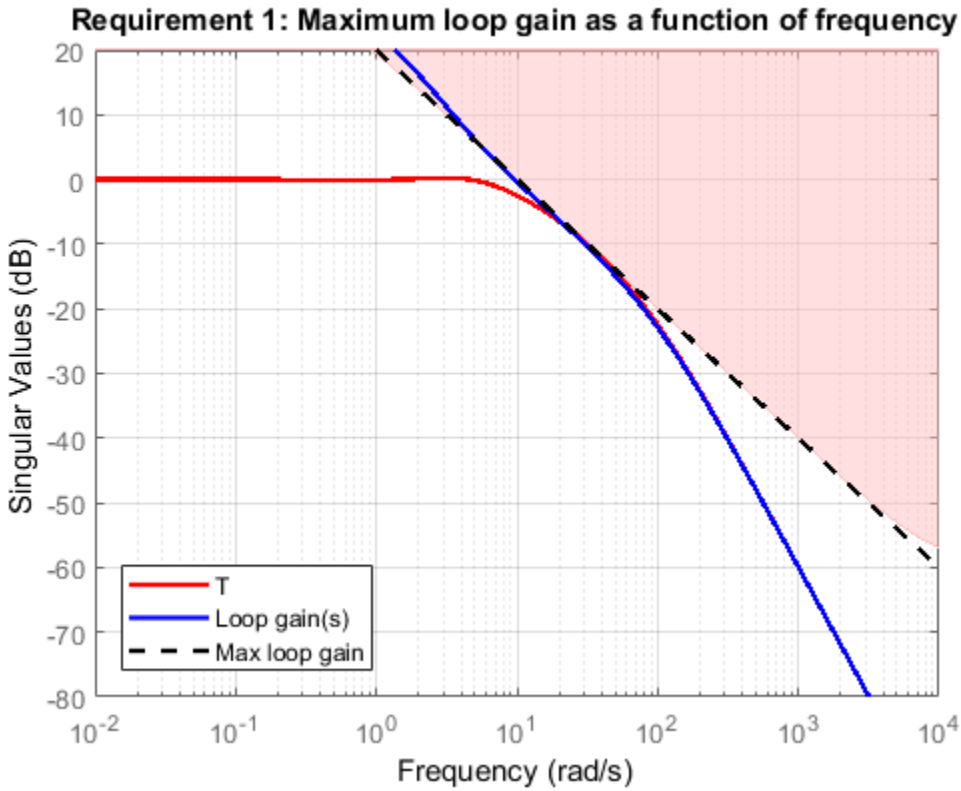
Description

Maximum Loop Gain Goal enforces a maximum loop gain in a particular frequency band. This tuning goal is useful, for example, for increasing system robustness to unmodeled dynamics.

Maximum Loop Gain Goal imposes a maximum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the maximum open-loop gain as a function of frequency (a maximum gain profile). For MIMO feedback loops, the specified gain profile is interpreted as an upper bound on the largest singular value of L.

When you tune a control system, the maximum gain profile is converted to a maximum gain constraint on the complementary sensitivity function, T) = $L/(I + L)$.

The following figure shows a typical specified maximum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The shaded region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much smaller than 1, imposing a maximum gain on T is a good proxy for a maximum open-loop gain.



Maximum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Maximum Loop Gain Goal and Minimum Loop Gain Goal specify only high-gain or low-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use “Loop Shape Goal” on page 9-140 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Maximum gain for open-loop response** to create a Maximum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MaxLoopGain` to specify a maximum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Desired Loop Gain

Use this section of the dialog box to specify the target maximum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target maximum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a target maximum gain and frequency. For example, to specify an integral gain profile

with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain below** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the maximum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the maximum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired maximum loop gain. For example, to specify maximum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the maximum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the maximum loop gain goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Maximum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_T \left(D^{-1} T D \right) \right\|_\infty .$$

Here, D is a diagonal scaling (for MIMO loops). T is the complementary sensitivity function at the specified location. W_T is a frequency-weighting function derived from the maximum loop gain profile you specify. The gain of this function roughly matches the inverse of the specified loop gain for values ranging from –60 dB to 20 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_T close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for

tuning, it is not recommended to specify gain profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see "Visualize Tuning Goals" on page 9-188.

Although T is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing an upper bound on the open-loop transfer, L, in a frequency band where the gain of L is less than one. To see why, note that $T = L/(I + L)$. For SISO loops, when $|L| << 1$, $|T| \approx |L|$. Therefore, enforcing the open-loop maximum gain requirement, $|L| < 1/|W_T|$, is roughly equivalent to enforcing $|W_T T| < 1$. For MIMO loops, similar reasoning applies, with $||T|| \approx \sigma_{\max}(L)$, where $\sigma_{\max}$ is the largest singular value.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40
- "Manage Tuning Goals" on page 9-177
- "Visualize Tuning Goals" on page 9-188

Loop Shape Goal

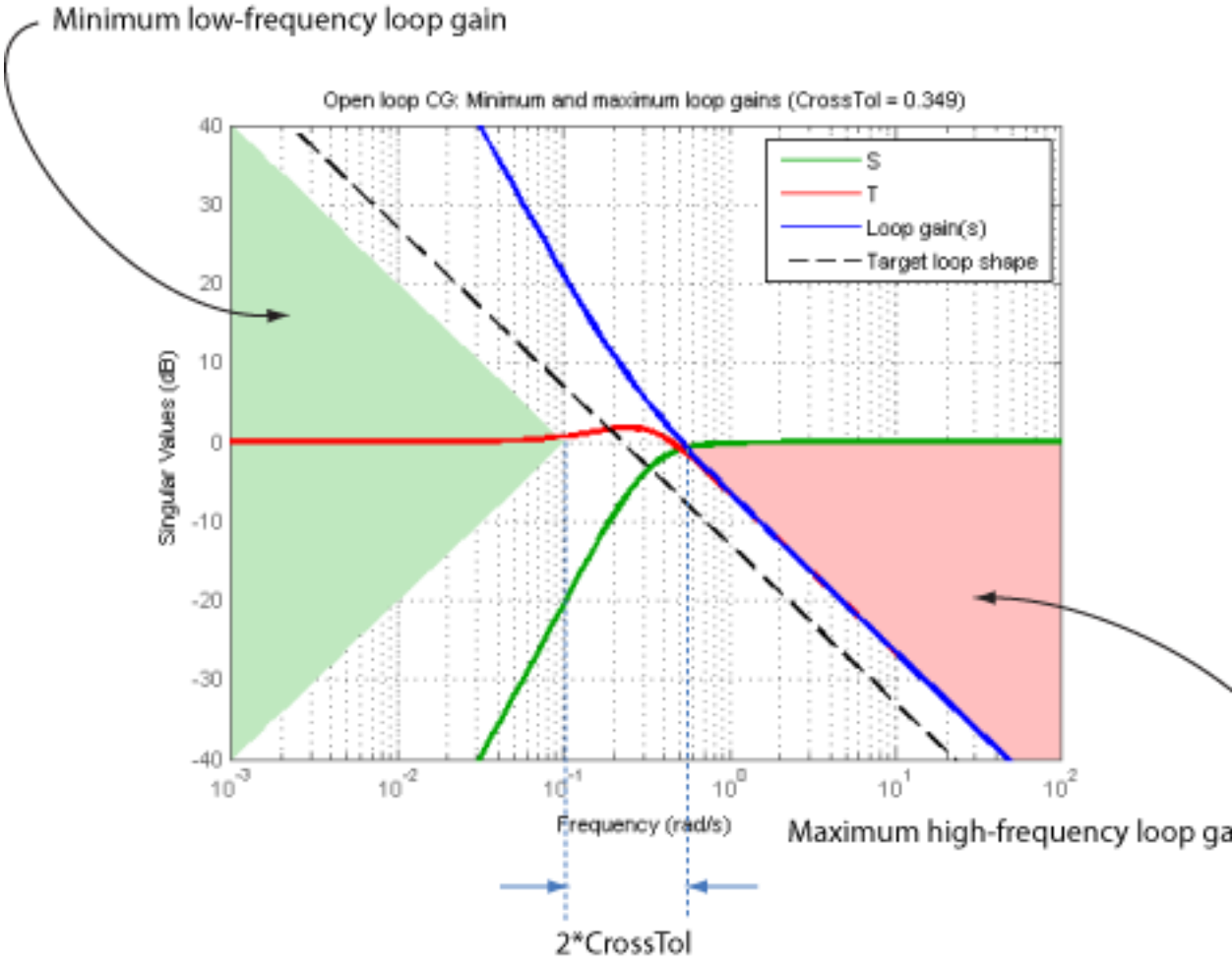
Purpose

Shape open-loop response of feedback loops when using Control System Tuner.

Description

Loop Shape Goal specifies a target gain profile (gain as a function of frequency) of an open-loop response. Loop Shape Goal constrains the open-loop, point-to-point response (L) at a specified location in your control system.

When you tune a control system, the target open-loop gain profile is converted into constraints on the inverse sensitivity function inv(S) = ($I + L$) and the complementary sensitivity function $T = 1–S$. These constraints are illustrated for a representative tuned system in the following figure.



Where L is much greater than 1, a minimum gain constraint on inv(S) (green shaded region) is equivalent to a minimum gain constraint on L. Similarly, where L is much smaller than 1, a maximum gain constraint on T (red shaded region) is equivalent to a maximum gain constraint on L. The gap between these two constraints is twice the crossover tolerance, which specifies the frequency band where the loop gain can cross 0 dB.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. Such values are lower

bounds on the smallest singular value of the open-loop response. Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of the open-loop response. For more information about singular values, see `sigma`.

Use Loop Shape Goal when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in certain frequency bands, use “Minimum Loop Gain Goal” on page 9-128 or “Maximum Loop Gain Goal” on page 9-134. When you do so, the software determines the best loop shape near crossover.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Target shape for open-loop response** to create a Loop Shape Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LoopShape` to specify a loop-shape goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Desired Loop Shape

Use this section of the dialog box to specify the target loop shape.

- **Pure integrator wc/s**

  Check to specify a pure integrator and crossover frequency for the target loop shape. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 10 in the **Crossover frequency wc** text box.

- **Other gain profile**

  Check to specify the target loop shape as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop shape using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired loop shape. For example, to specify a target loop shape of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/decade at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the loop shape as a discrete-time model with the same sample time that you are using for tuning. If you specify the loop shape in continuous time, the tuning software discretizes it. Specifying the loop shape in discrete time gives you more control over the loop shape near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the loop shape goal.

- **Enforce loop shape within**

  Specify the tolerance in the location of the crossover frequency, in decades. For example, to allow gain crossovers within half a decade on either side of the target crossover frequency, enter 0.5. Increase the crossover tolerance to increase the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

- **Enforce goal in frequency range**

Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the

control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Loop Shape Goal**, $f(x)$ is given by:

$$f(x) = \left\| \begin{matrix} W_S S \\ W_T T \end{matrix} \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. (If **Equalize loop interactions** is set to `Off`, then $D = I$.)

$T = S - I$ is the complementary sensitivity function.

W_S and W_T are frequency weighting functions derived from the specified loop shape. The gains of these functions roughly match your specified loop shape and its inverse, respectively, for values ranging from –20 dB to 60 dB. For numerical reasons, the weighting functions level off outside this range, unless the specified gain profile changes slope outside this range. Because poles of W_S or W_T close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify loop shapes with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 9-188.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

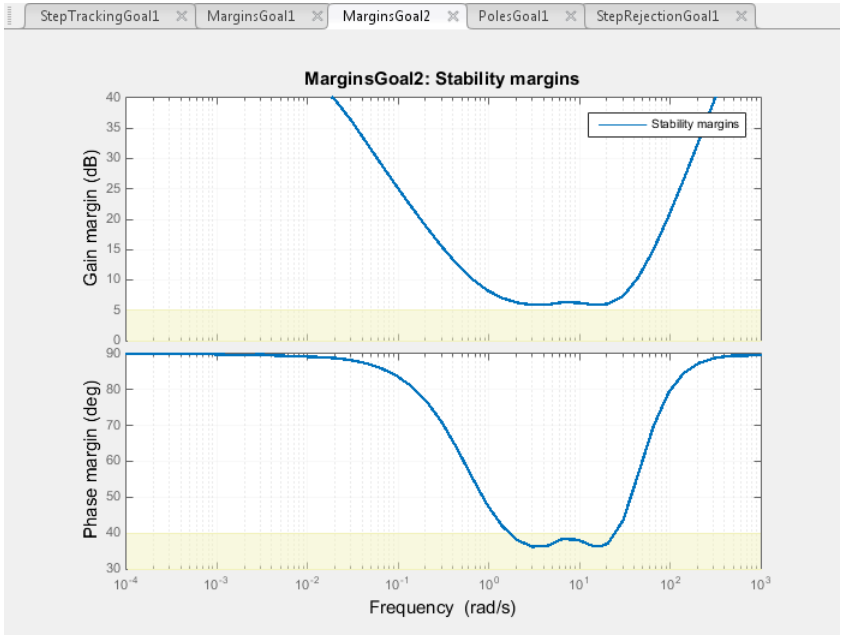
Margins Goal

Purpose

Enforce specified gain and phase margins when using Control System Tuner.

Description

Margins Goal enforces specified gain and phase margins on a SISO or MIMO feedback loop. For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the margins goal is not met. For more information about interpreting this plot, see “Stability Margins in Control System Tuning” on page 9-217.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Minimum stability margins** to create a Margins Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Margins` to specify a stability margin goal.

Feedback Loop Selection

Use this section of the dialog box to specify the signal locations at which to measure stability margins. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Measure stability margins at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the stability margins. To constrain a SISO loop, select a single-valued location. For example, to constrain the stability margins at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO loop, select multiple signals or a vector-valued signal.

- **Measure stability margins with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Desired Margins

Use this section of the dialog box to specify the minimum gain and phase margins for the feedback loop.

- **Gain margin (dB)**

  Enter the required minimum gain margin for the feedback loop as a scalar value expressed in dB.

- **Phase margin (degrees)**

  Enter the required minimum phase margin for the feedback loop as a scalar value expressed in degrees.

For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.

Options

Use this section of the dialog box to specify additional characteristics of the stability margin goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies.

- **D scaling order**

  This value controls the order (number of states) of the scalings involved in computing MIMO stability margins. Static scalings (scaling order 0) are used by default. Increasing the order may improve results at the expense of increased computations. If the stability margin plot shows a large gap between the optimized and actual margins, consider increasing the scaling order. See “Stability Margins in Control System Tuning” on page 9-217.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth

models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Margins Goal**, $f(x)$ is given by:

$$f(x) = \left\| 2\alpha S - \alpha I \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor.

α is a scalar parameter computed from the specified gain and phase margin.

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40

- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

More About

- “Stability Margins in Control System Tuning” on page 9-217

Passivity Goal

Purpose

Enforce passivity of specific input/output map when using Control System Tuner.

Description

Passivity Goal enforces passivity of the response of the transfer function between the specified signal locations. A system is passive if all its I/O trajectories ($u(t)$,$y(t)$) satisfy:

$$\int_0^T y(t)^{\mathrm{T}} u(t)\, dt > 0,$$

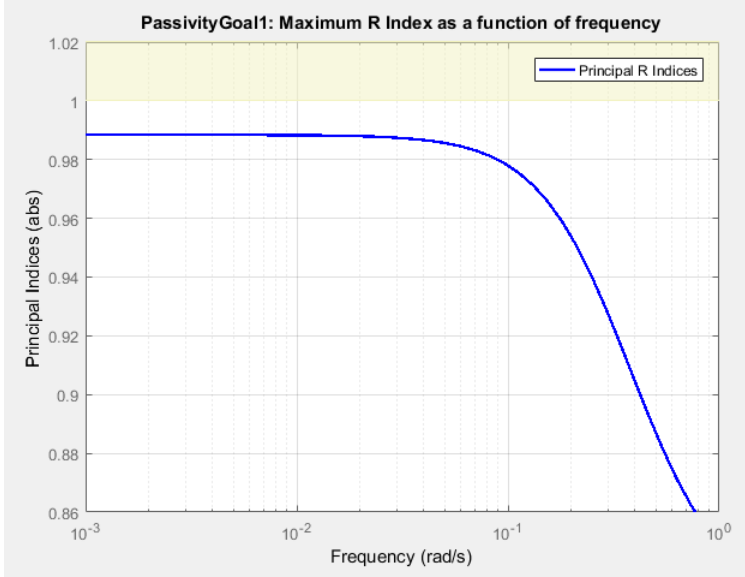
for all $T > 0$. Equivalently, a system is passive if its frequency response is positive real, which means that for all $\omega > 0$,

$$G(j\omega) + G(j\omega)^H > 0$$

Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^{\mathrm{T}} u(t)\, dt > \nu \int_0^T u(t)^{\mathrm{T}} u(t) dt + \rho \int_0^T y(t)^{\mathrm{T}} y(t) dt,$$

for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (ν) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs, set ν or ρ to a positive value. To permit a shortage of passivity, set ν or ρ to a negative value. See “About Passivity and Passivity Indices” (Control System Toolbox) for more information about these indices.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in "Algorithms" on page 9-155.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Passivity Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Passivity` to specify a passivity constraint.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued

input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Options

Use this section of the dialog box to specify additional characteristics of the passivity goal.

- **Minimum input passivity index**

  Enter the target value of ν in the text box. To enforce an excess of passivity at the specified inputs, set $\nu > 0$. To permit a shortage of passivity, set $\nu < 0$. By default, the passivity goal enforces $\nu = 0$, passive at the inputs with no required excess of passivity.

- **Minimum output passivity index**

  Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. To permit a shortage of passivity, set $\rho < 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units

of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Passivity Goal**, for a closed-loop transfer function $G(s,x)$ from the specified inputs to the specified outputs, $f(x)$ is given by:

$$f\left(x\right)=\frac{R}{1+R/R_{\max}},\quad R_{\max}=10^{6}.$$

R is the relative sector index (see `getSectorIndex`) of $[G(s,x); I]$, for the sector represented by:

$$Q=\begin{pmatrix}2\rho & -I\\ -I & 2\nu\end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. $R_{\max}$ is fixed at 10_6, included to avoid numeric errors for very large R.

This tuning goal imposes an implicit minimum-phase constraint on the transfer function $G + I$. The transmission zeros of $G + I$ are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 9-40
- "Manage Tuning Goals" on page 9-177
- "Visualize Tuning Goals" on page 9-188
- "Passive Control of Water Tank Level" (Control System Toolbox)
- "About Passivity and Passivity Indices" (Control System Toolbox)

Conic Sector Goal

Purpose

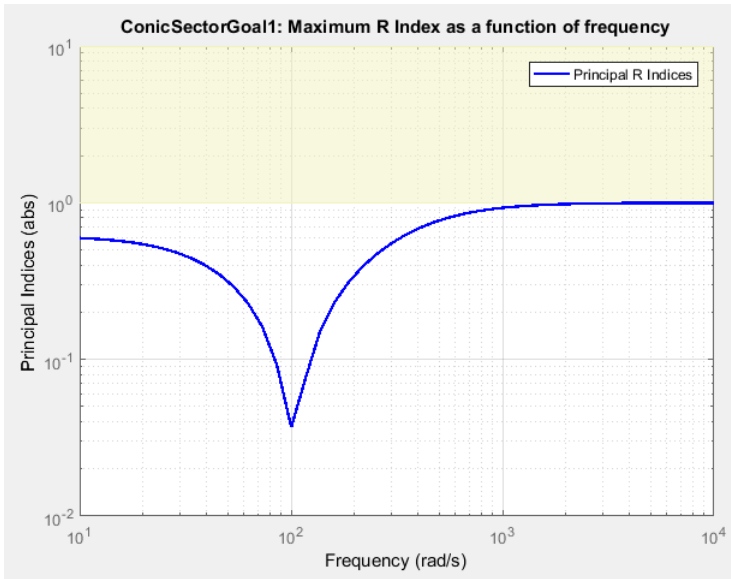
Enforce sector bound on specific input/output map when using Control System Tuner.

Description

Conic Sector Goal creates a constraint that restricts the output trajectories of a system. If for all nonzero input trajectories $u(t)$, the output trajectory $z(t) = (Hu)(t)$ of a linear system H satisfies:

$$\int_0^T z(t)^{\mathrm{T}} Q\, z(t) dt < 0,$$

for all $T \geq 0$, then the output trajectories of H lie in the conic sector described by the symmetric indefinite matrix Q. Selecting different Q matrices imposes different conditions on the system response. When you create a Conic Sector Goal, you specify the input signals, output signals, and the sector geometry.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the *R*-index described in "About Sector Bounds and Sector Indices" (Control System Toolbox).

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Conic Sector Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.ConicSector` to specify a step response goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Options

Specify additional characteristics of the conic sector goal using this section of the dialog box.

- **Conic Sector Matrix**

  Enter the sector geometry Q, specified as:

  - A matrix, for constant sector geometry. `Q` is a symmetric square matrix that is `ny` on a side, where `ny` is the number of output signals you specify for the goal. The matrix `Q` must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues. In particular, `Q` must have as many negative eigenvalues as there are input signals specified for the tuning goal (the size of the vector input signal $u(t)$).
  - An LTI model, for frequency-dependent sector geometry. `Q` satisfies $Q(s)' = Q(-s)$. In other words, $Q(s)$ evaluates to a Hermitian matrix at each frequency.

  For more information, see "About Sector Bounds and Sector Indices" (Control System Toolbox).

- **Regularization**

  Regularization parameter, specified as a real nonnegative scalar value. Regularization keeps the evaluation of the tuning goal numerically tractable when other tuning goals tend to make the sector bound ill-conditioned at some frequencies. When this condition occurs, set **Regularization** to a small (but not negligible) fraction of the typical norm of the feedthrough term in H. For example, if you anticipate the norm of the feedthrough term of H to be of order 1 during tuning, try setting **Regularization** to 0.001.

  For more information about the conditions that require regularization, see the `Regularization` property of `TuningGoal.ConicSector`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

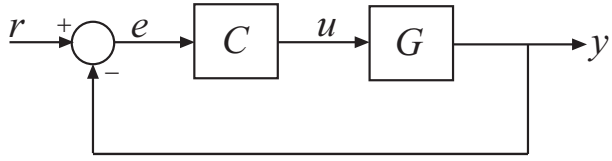
  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Tips

Constraining Input and Output Trajectories to Conic Sector

Consider the following control system.



Suppose that the signal u is marked as an analysis point in the model you are tuning. Suppose also that G is the closed-loop transfer function from u to y. A common application is to create a tuning goal that constrains all the I/O trajectories $\{u(t),y(t)\}$ of G to satisfy:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^{\mathrm{T}} Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0,$$

for all $T \geq 0$. Constraining the I/O trajectories of G is equivalent to restricting the output trajectories $z(t)$ of the system $H = [G;I]$ to the sector defined by:

$$\int_0^T z(t)^{\mathrm{T}} Q\, z(t)\, dt < 0.$$

(See "About Sector Bounds and Sector Indices" (Control System Toolbox) for more details about this equivalence.) To specify a constraint of this type using Conic Sector Goal, specify u as the input signal, and specify y and u as output signals. When you specify u as both input and output, Conic Sector Goal sets the corresponding transfer function to the identity. Therefore, the transfer function that the goal constrains is $H = [G;I]$ as intended. This treatment is specific to Conic Sector Goal. For other tuning goals, when the same signal appears in both inputs and outputs, the resulting transfer function is zero in the absence of feedback loops, or the complementary sensitivity at that location otherwise. This result occurs because when the software processes analysis points, it assumes that the input is injected after the output. See "Mark Signals of Interest for Control System Analysis and Design" on page 2-48 for more information about how analysis points work.

Algorithms

Let

$$Q = W_1 W_1^{\mathrm{T}} - W_2 W_2^{\mathrm{T}}$$

be an indefinite factorization of Q, where $W_1^{\mathrm{T}} W_2 = 0$. If $W_2^{\mathrm{T}} H(s)$ is square and minimum phase, then the time-domain sector bound

$$\int_0^T z(t)^{\mathrm{T}}\, Q\, z(t)\, dt < 0,$$

is equivalent to the frequency-domain sector condition,

$$H(-j\omega) Q H(j\omega) < 0$$

for all frequencies. Conic Sector Goal uses this equivalence to convert the time-domain characterization into a frequency-domain condition that Control System Tuner can handle in the same way it handles gain constraints. To secure this equivalence, Conic

Sector Goal also makes $W_2^{\mathrm{T}} H(s)$ minimum phase by making all its zeros stable. The transmission zeros affected by this minimum-phase condition are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

For sector bounds, the R-index plays the same role as the peak gain does for gain constraints (see "About Sector Bounds and Sector Indices" (Control System Toolbox)). The condition

$$H(-j\omega)QH(j\omega) < 0$$

is satisfied at all frequencies if and only if the R-index is less than one. The plot that Control System Tuner displays for Conic Sector Goal shows the R-index value as a function of frequency (see `sectorplot`).

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$, where x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For Conic Sector Goal, for a closed-loop transfer function `H(s,x)` from the specified inputs to the specified outputs, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of `H(s,x)`, for the sector represented by `Q`.

See Also

Related Examples

- "About Sector Bounds and Sector Indices" (Control System Toolbox)
- "Specify Goals for Interactive Tuning" on page 9-40
- "Manage Tuning Goals" on page 9-177
- "Visualize Tuning Goals" on page 9-188

Weighted Passivity Goal

Purpose

Enforce passivity of a frequency-weighted transfer function when tuning in Control System Tuner.

Description

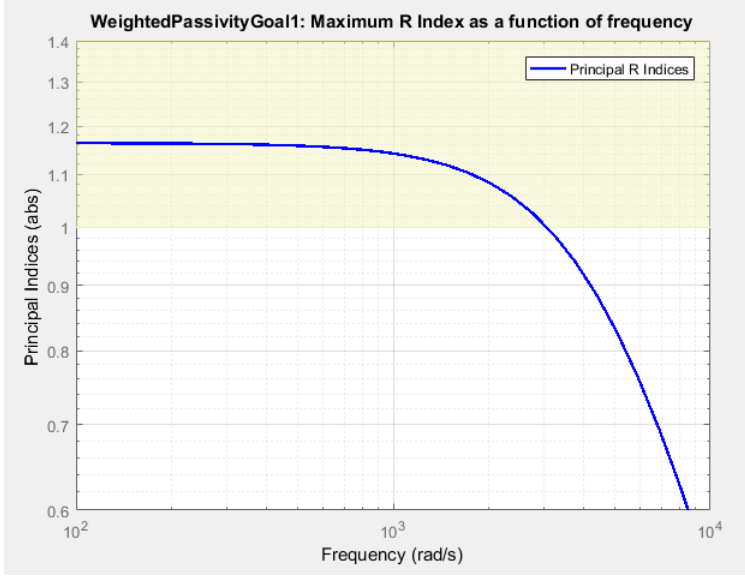
Weighted Passivity Goal enforces the passivity of $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs. $W_L(s)$ and $W_R(s)$ are frequency weights used to emphasize particular frequency bands. A system is passive if all its I/O trajectories $(u(t),y(t))$ satisfy:

$$\int_0^T y(t)^{\mathrm{T}} u(t)\, dt > 0,$$

for all $T > 0$. Weighted Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^{\mathrm{T}} u(t)\, dt > \nu \int_0^T u(t)^{\mathrm{T}} u(t) dt + \rho \int_0^T y(t)^{\mathrm{T}} y(t) dt,$$

for the trajectories of the weighted transfer function $H(s)$, for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (ν) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs of the weighted transfer function, set ν or ρ to a positive value. To permit a shortage of passivity, set ν or ρ to a negative value. See `getPassiveIndex` for more information about these indices.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in “Algorithms” on page 9-167.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Weighted Passivity Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedPassivity` to specify a step response goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs.

W_L provides the weighting for the output channels of $H(s)$, and W_R provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying matrices or MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of W_L and W_R. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the step response goal.

- **Minimum input passivity index**

  Enter the target value of ν in the text box. To enforce an excess of passivity at the specified inputs, set $\nu > 0$. To permit a shortage of passivity, set $\nu < 0$. By default, the passivity goal enforces $\nu = 0$, passive at the inputs with no required excess of passivity.

- **Minimum output passivity index**

  Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. To permit a shortage of passivity, set $\rho < 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Passivity Goal**, for a closed-loop transfer function $T(s,x)$ from the specified inputs to the specified outputs, and the weighted transfer function $H(s,x) = W_L(s)T(s,x)W_R(s)$, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of $[H(s,x); I]$, for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. $R_{\max}$ is fixed at 10_6, included to avoid numeric errors for very large R.

This tuning goal imposes an implicit minimum-phase constraint on the weighted transfer function $H + I$. The transmission zeros of $H + I$ are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188
- “About Passivity and Passivity Indices” (Control System Toolbox)

Poles Goal

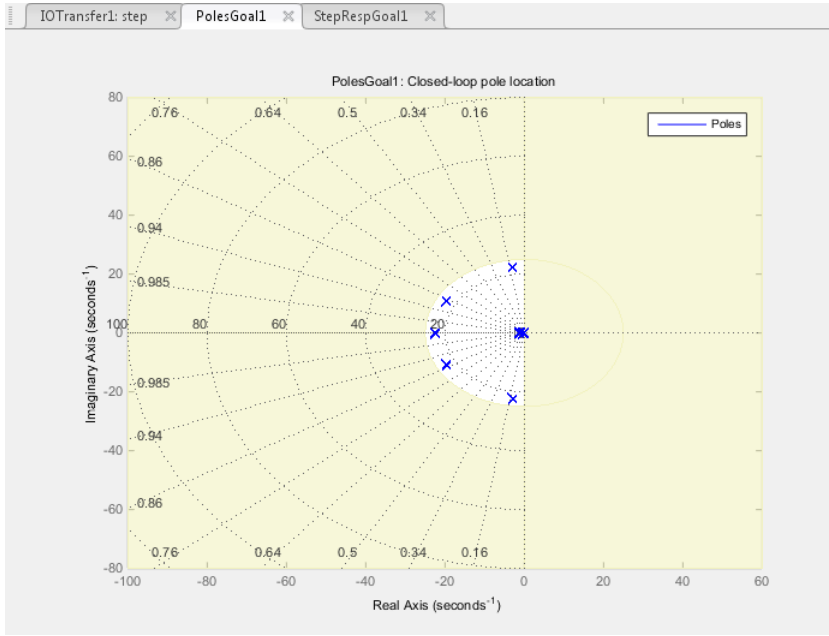
Purpose

Constrain the dynamics of the closed-loop system, specified feedback loops, or specified open-loop configurations, when using Control System Tuner.

Description

Poles Goal constrains the dynamics of your entire control system or of specified feedback loops of your control system. Constraining the dynamics of a feedback loop means constraining the dynamics of the sensitivity function measured at a specified location in the control system.

Using Poles Goal, you can specify finite minimum decay rate or minimum damping for the poles in the control system or specified loop. You can specify a maximum natural frequency for these poles, to eliminate fast dynamics in the tuned control system.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met.

To constrain dynamics or ensure stability of a single tunable component of the control system, use “Controller Poles Goal” on page 9-174.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Constraint on closed-loop dynamics** to create a Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Poles` to specify a disturbance rejection goal.

Feedback Configuration

Use this section of the dialog box to specify the portion of the control system for which you want to constrain dynamics. You can also specify loop-opening locations for evaluating the tuning goal.

- **Entire system**

  Select this option to constrain the locations of closed-loop poles of the control system.

- **Specific feedback loop(s)**

  Select this option to specify one or more feedback loops to constrain. Specify a feedback loop by selecting a signal location in your control system. Poles Goal constrains the dynamics of the sensitivity function measured at that location. (See `getSensitivity` for information about sensitivity functions.)

  To constrain the dynamics of a SISO loop, select a single-valued location. For example, to constrain the dynamics of the sensitivity function measured at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the dynamics of a MIMO loop, select multiple signals or a vector-valued signal.

- **Compute poles with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the

open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Pole Location

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the target minimum decay rate for the system poles. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDecay` for continuous-time systems, or `log(|z|) < -MinDecay*Ts` for discrete-time systems with sample time `Ts`. This constraint helps ensure stable dynamics in the tuned system.

  Enter 0 to impose no constraint on the decay rate.

- **Minimum damping**

  Enter the target minimum damping of closed-loop poles of tuned system, as a value between 0 and 1. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s = log(z)/Ts`.

  Enter 0 to impose no constraint on the damping ratio.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of tuned system, in the units of the control system model you are tuning. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `|s| < MaxFrequency` for continuous-time systems, or `|log(z)| < MaxFrequency*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the control system.

  Enter `Inf` to impose no constraint on the natural frequency.

Options

Use this section of the dialog box to specify additional characteristics of the poles goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  The Poles Goal applies only to poles with natural frequency within the range you specify.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see "Robust Tuning Approaches" (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Poles Goal constrains the closed-loop poles of a feedback loop to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the smallest damping among the constrained poles is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the smallest damping $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the smallest damping $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

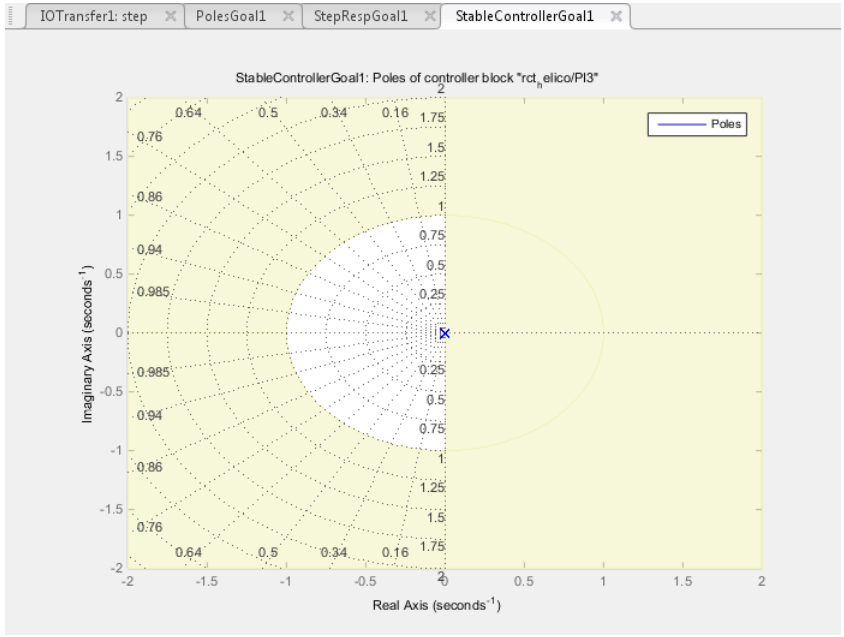
Controller Poles Goal

Purpose

Constrain the dynamics of a specified tunable block in the tuned control system, when using Control System Tuner.

Description

Controller Poles Goal constrains the dynamics of a tunable block in your control system model. Controller Poles Goal can impose a stability constraint on the specified block. You can also specify a finite minimum decay rate, a minimum damping rate, or a maximum natural frequency for the poles of the block. These constraints allow you to eliminate fast dynamics and control ringing in the response of the tunable block.



In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met. The constraint applies to all poles in the block except fixed integrators, such as the I term of a PID controller.

To constrain dynamics or ensure stability of an entire control system or a feedback loop in the control system, use “Poles Goal” on page 9-169.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Constraint on controller dynamics** to create a Controller Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.ControllerPoles` to specify a controller poles goal.

Constrain Dynamics of Tuned Block

From the drop-down menu, select the tuned block in your control system to which to apply the Controller Poles Goal.

If the block you want to constrain is not in the list, add it to the Tuned Blocks list. In Control System Tuner, in the **Tuning** tab, click  **Select Blocks**. For more information about adding tuned blocks, see “Specify Blocks to Tune in Control System Tuner” on page 9-25.

Keep Poles Inside the Following Region

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the desired minimum decay rate for the poles of the tunable block. Poles of the block are constrained to satisfy `Re(s) < -MinDecay` for continuous-time blocks, or `log(|z|) < -MinDecay*Ts` for discrete-time blocks with sample time `Ts`.

  Specify a nonnegative value to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

- **Minimum damping**

  Enter the desired minimum damping ratio of poles of the tunable block, as a value between 0 and 1. Poles of the block that depend on the tunable parameters are

constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s=log(z)/Ts`.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of the tunable block, in the units of the control system model you are tuning. Poles of the block are constrained to satisfy `|s| < MaxFrequency` for continuous-time blocks, or `|log(z)| < MaxFrequency*Ts` for discrete-time blocks with sample time `Ts`. This constraint prevents fast dynamics in the tunable block.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Controller Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Controller Poles Goal constrains the pole of a tuned block to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the damping of the pole is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the damping is $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the damping is $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 9-40
- “Manage Tuning Goals” on page 9-177
- “Visualize Tuning Goals” on page 9-188

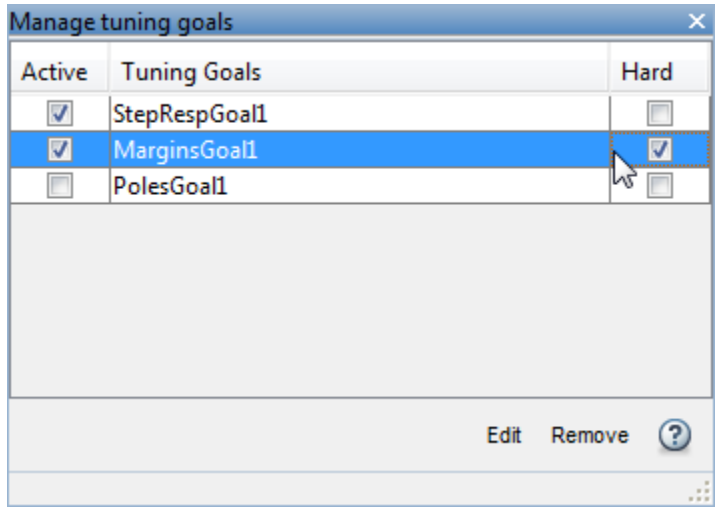
Manage Tuning Goals

Control System Tuner lets you designate one or more tuning goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have goals. Control System Tuner attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

By default, new goals are designated soft goals. In the **Tuning** tab, click  **Manage Goals** to open the **Manage tuning goals** dialog box. Check **Hard** for any goal to designate it a hard goal.

You can also designate any tuning goal as inactive for tuning. In this case the software ignores the tuning goal entirely. Use this dialog box to select which tuning goals are active when you tune the control system. **Active** is checked by default for any new goals. Uncheck **Active** for any design goal that you do not want enforced.

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.



All tuning goals you have created in the Control System Tuner session are listed in the dialog box. To edit an existing tuning goal, select it in the list and click **Edit**. To delete a tuning goal from the list, select it and click **Remove**.

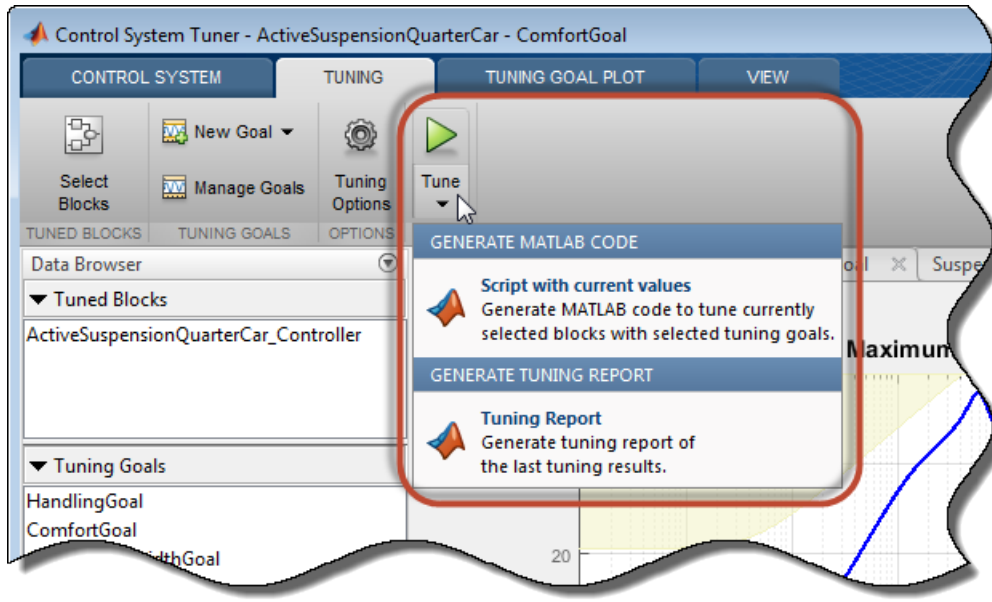
To add more tuning goals to the list, in Control System Tuner, in the **Tuning** tab, click **New Goal**. For more information about creating tuning goals, see “Specify Goals for Interactive Tuning” on page 9-40.

Generate MATLAB Code from Control System Tuner for Command-Line Tuning

You can generate a MATLAB script in Control System Tuner for tuning a control system at the command line. Generated scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB script also enables you to programmatically perform multiple tuning operations with variations in tuning goals, system parameters, or model conditions such as operating point.

Tip You can also save a Control System Tuner session to reproduce within Control System Tuner. To do so, in the **Control System** tab, click **Save Session**.

To generate a MATLAB script in Control System Tuner, in the **Tuning** tab, click **Tune** ▾. Select **Script with current values**.



The MATLAB Editor displays the generated script, which script reproduces programmatically the current tuning configuration of Control System Tuner.

For example, suppose you generate a MATLAB script after completing all steps in the example "Control of a Linear Electric Actuator Using Control System Tuner" (Control System Toolbox). The generated script computes the operating point used for tuning, designates the blocks to tune, creates the tuning goals, and performs other operations to reproduce the result at the command line.

The first section of the script creates the `slTuner` interface to the Simulinkmodel (`rct_linact` in this example). The `slTuner` interface stores a linearization of the model and parameterizations of the blocks to tune.

```
%% Create system data with slTuner interface
TunedBlocks = {'rct_linact/Current Controller/Current PID'; ...
               'rct_linact/Speed Controller/Speed PID'};
AnalysisPoints = {'rct_linact/Speed Demand (rpm)/1'; ...
                  'rct_linact/Current Sensor/1'; ...
                  'rct_linact/Hall Effect Sensor/1'; ...
                  'rct_linact/Speed Controller/Speed PID/1'; ...
                  'rct_linact/Current Controller/Current PID/1'};
OperatingPoints = 0.5;
% Specify the custom options
Options = slTunerOptions('AreParamsTunable',false);
% Create the slTuner object
CL0 = slTuner('rct_linact',TunedBlocks,AnalysisPoints,OperatingPoints,Options);
```

The `slTuner` interface also specifies the operating point at which the model is linearized, and marks as analysis points all the signal locations required to specify the tuning goals for the example. (See "Create and Configure slTuner Interface to Simulink Model" on page 9-211.)

If you are tuning a control system modeled in MATLAB instead of Simulink, the first section of the script constructs a `genss` model that has equivalent dynamics and parameterization to the `genss` model of the control system that you specified Control System Tuner.

Next, the script creates the three tuning goals specified in the example. The script uses `TuningGoal` objects to capture these tuning goals. For instance, the script uses `TuningGoal.Tracking` to capture the Tracking Goal of the example.

```
%% Create tuning goal to follow reference commands with prescribed performance
% Inputs and outputs
Inputs = {'rct_linact/Speed Demand (rpm)/1'};
Outputs = {'rct_linact/Hall Effect Sensor/1[rpm]'};
% Tuning goal specifications
ResponseTime = 0.1; % Approximately reciprocal of tracking bandwidth
```

```
DCError = 0.001; % Maximum steady-state error
PeakError = 1; % Peak error across frequency
% Create tuning goal for tracking
TR = TuningGoal.Tracking(Inputs,Outputs,ResponseTime,DCError,PeakError);
TR.Name = 'TR'; % Tuning goal name
```

After creating the tuning goals, the script sets any algorithm options you had set in Control System Tuner. The script also designates tuning goals as soft or hard goals, according to the configuration of tuning goals in Control System Tuner. (See "Manage Tuning Goals" on page 9-177.)

```
%% Create option set for systune command
Options = systuneOptions();

%% Set soft and hard goals
SoftGoals = [ TR ; ...
              MG1 ; ...
              MG2 ];
HardGoals = [];
```

In this example, all the goals are designated as soft goals when the script is generated. Therefore, `HardGoals` is empty.

Finally, the script tunes the control system by calling `systune` on the `slTuner` interface using the tuning goals and options.

```
%% Tune the parameters with soft and hard goals
[CL1,fSoft,gHard,Info] = systune(CL0,SoftGoals,HardGoals,Options);
```

The script also includes an optional call to `viewGoal`, which displays graphical representations of the tuning goals to aid you in interpreting and validating the tuning results. Uncomment this line of code to generate the plots.

```
%% View tuning results
% viewGoal([SoftGoals;HardGoals],CL1);
```

You can add calls to functions such `getIOTransfer` to make the script generate additional analysis plots.

See Also

Related Examples

- “Create and Configure slTuner Interface to Simulink Model” on page 9-211
- “Tune Control System at the Command Line” on page 9-222
- “Validate Tuned Control System” on page 9-226

Interpret Numeric Tuning Results

When you tune a control system with `systune` or Control System Tuner, the software provides reports that give you an overview of how well the tuned control system meets your design requirements. Interpreting these reports requires understanding how the tuning algorithm optimizes the system to satisfy your tuning goals. (The software also provides visualizations of the tuning goals and system responses to help you see where and by how much your requirements are not satisfied. For information about using these plots, see "Visualize Tuning Goals" on page 9-188.)

Tuning-Goal Scalar Values

The tuning software converts each tuning goal into a normalized scalar value which it then constrains (hard goals) or minimizes (soft goals). Let $f_i(x)$ and $g_j(x)$ denote the scalar values of the soft and hard goals, respectively. Here, x is the vector of tunable parameters in the control system to tune. The tuning algorithm solves the minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system. (For information about the specific functions used to evaluate each type of requirement, see the reference pages for each tuning goal.)

When you use both soft and hard tuning goals, the software solves the optimization as a sequence of subproblems of the form:

$$\min_x \ \max\left(\alpha f(x), g(x)\right).$$

The software adjusts the multiplier α so that the solution of the subproblems converges to the solution of the original constrained optimization problem.

The tuning software reports the final scalar values for each tuning goal. When the final value of $f_i(x)$ or $g_j(x)$ is less than 1, the corresponding tuning goal is satisfied. Values greater than 1 indicate that the tuning goal is not satisfied for at least some conditions. For instance, a tuning goal that describes a frequency-domain constraint might be satisfied at some frequencies and not at others. The closer the value is to 1, the closer the tuning goal is to being satisfied. Thus these values give you an overview of how successfully the tuned system meets your requirements.

The form in which the software presents the optimized tuning-goal values depends on whether you are tuning with Control System Tuner or at the command line.

Tuning Results at the Command Line

The `systune` command returns the control system model or `slTuner` interface with the tuned parameter values. `systune` also returns the best achieved values of each $f_i(x)$ and $g_j(x)$ as the vector-valued output arguments `fSoft` and `gHard`, respectively. See the `systune` reference page for more information. (To obtain the final tuning goal values on their own, use `evalGoal`.)

By default, `systune` displays the best achieved final values of the tuning goals in the command window. For instance, in the example "PID Tuning for Setpoint Tracking vs. Disturbance Rejection" (Control System Toolbox), `systune` is called with one soft requirement, `R1`, and two hard requirements `R2` and `R3`.

```
T1 = systune(T0,R1,[R2 R3]);
```

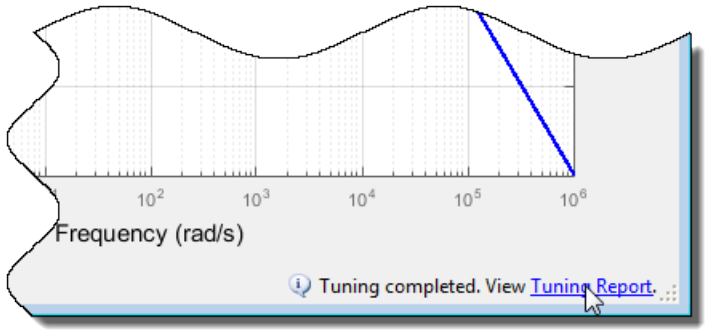
```
Final: Soft = 1.12, Hard = 0.99988, Iterations = 143
```

This display indicates that the largest optimized value of the hard tuning goals is less than 1, so both hard goals are satisfied. The soft goal value is slightly greater than one, indicating that the soft goal is nearly satisfied. You can use tuning-goal plots to see in what regimes and by how much the tuning goals are violated. (See "Visualize Tuning Goals" on page 9-188.)

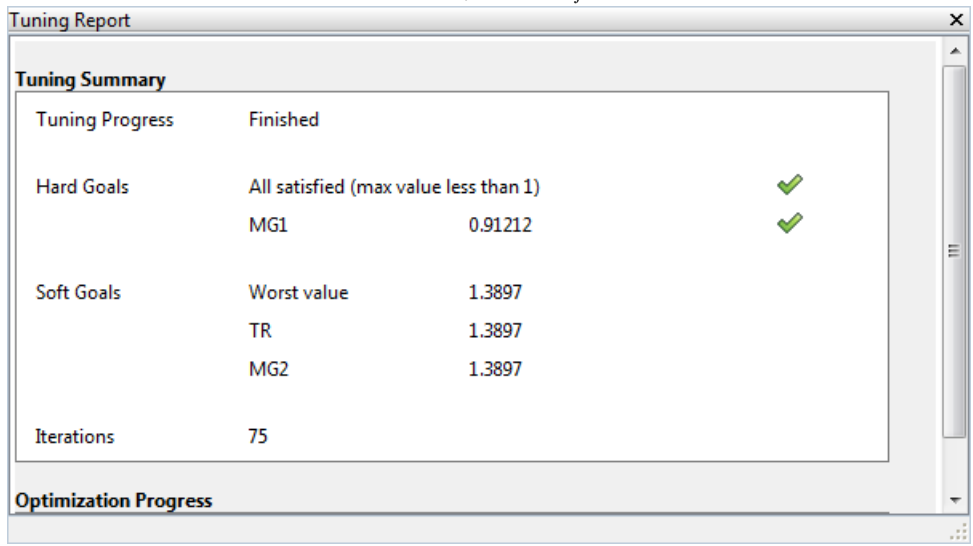
You can obtain additional information about the optimization progress and values using the `info` output of `systune`. To make `systune` display additional information during tuning, use `systuneOptions`.

Tuning Results in Control System Tuner

In Control System Tuner, when you click  , the app compiles a Tuning Report summarizing the best achieved values of $f_i(x)$ and $g_j(x)$. To view the tuning report immediately after tuning a control system, click **Tuning Report** at the bottom-right corner of Control System Tuner.

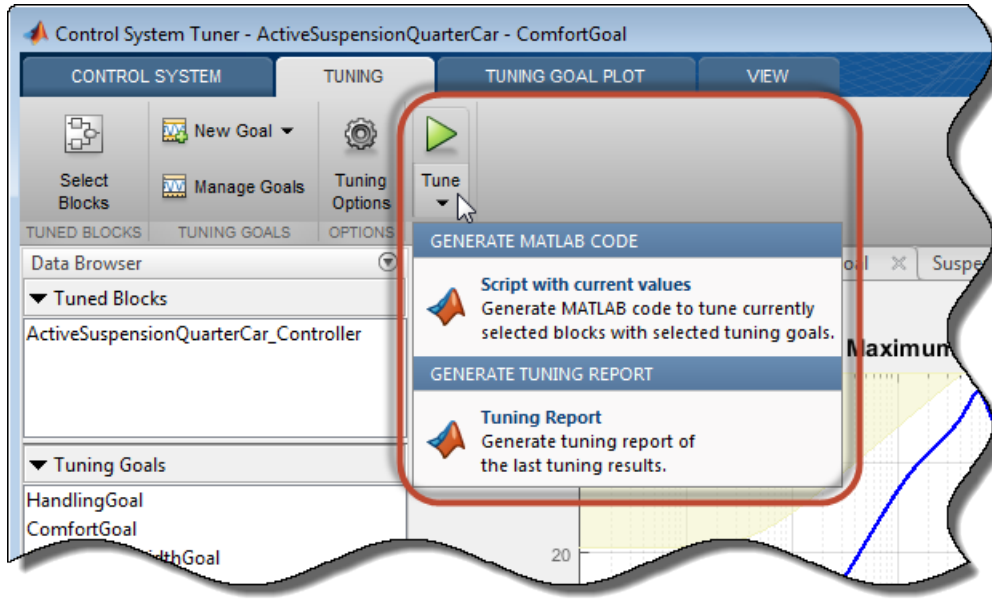


The tuning report displays the final $f_i(x)$ and $g_j(x)$ values obtained by the algorithm.



The **Hard Goals** area shows the minimized $g_i(x)$ values and indicates which are satisfied. The **Soft Goals** area highlights the largest of the minimized $f_i(x)$ values as **Worst Value**, and lists the values for all the requirements. In this example, the hard goal is satisfied, while the soft goals are nearly satisfied. As in the command-line case, you can use tuning-goal plots to see where and by how much tuning goals are violated. (See “Visualize Tuning Goals” on page 9-188.)

Tip You can view a report from the most recent tuning run at any time. In the **Tuning** tab, click **Tune**, and select `Tuning Report`.



Improve Tuning Results

If the tuning results do not adequately meet your design requirements, adjust your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced.
  - In Control System Tuner, use the **Enforce goal in frequency range** field of the tuning goal dialog box.
  - At the command line, use the `Focus` property of the `TuningGoal` object.

If the tuning results do satisfy your design requirements, you can validate the tuned control system as described in “Validate Tuned Control System” on page 9-226.

See Also

`evalGoal` | `systune` | `systune (for slTuner)` | `viewGoal`

Related Examples

- "Visualize Tuning Goals" on page 9-188
- "Validate Tuned Control System" on page 9-226

Visualize Tuning Goals

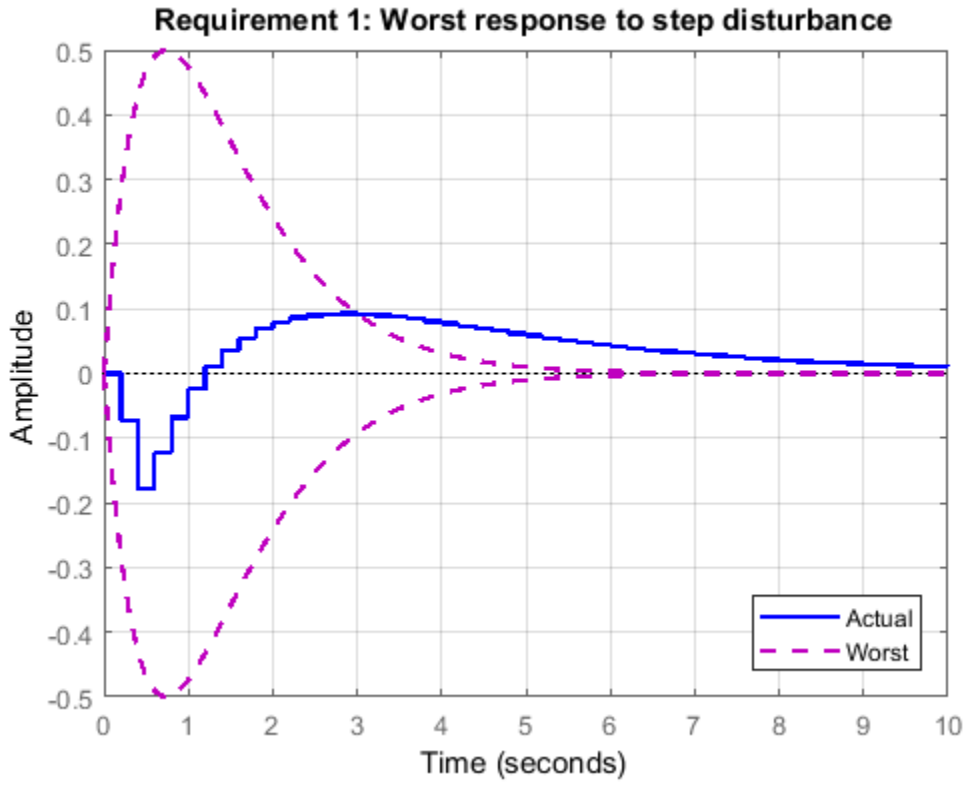
When you tune a control system with `systune` or Control System Tuner, use tuning-goal plots to visualize your design requirements against the tuned control system responses. Tuning-goal plots show graphically where and by how much tuning goals are satisfied or violated. This visualization lets you examine how close your control system is to ideal performance. It can also help you identify problems with tuning and provide clues on how to improve your design.

Tuning-Goal Plots

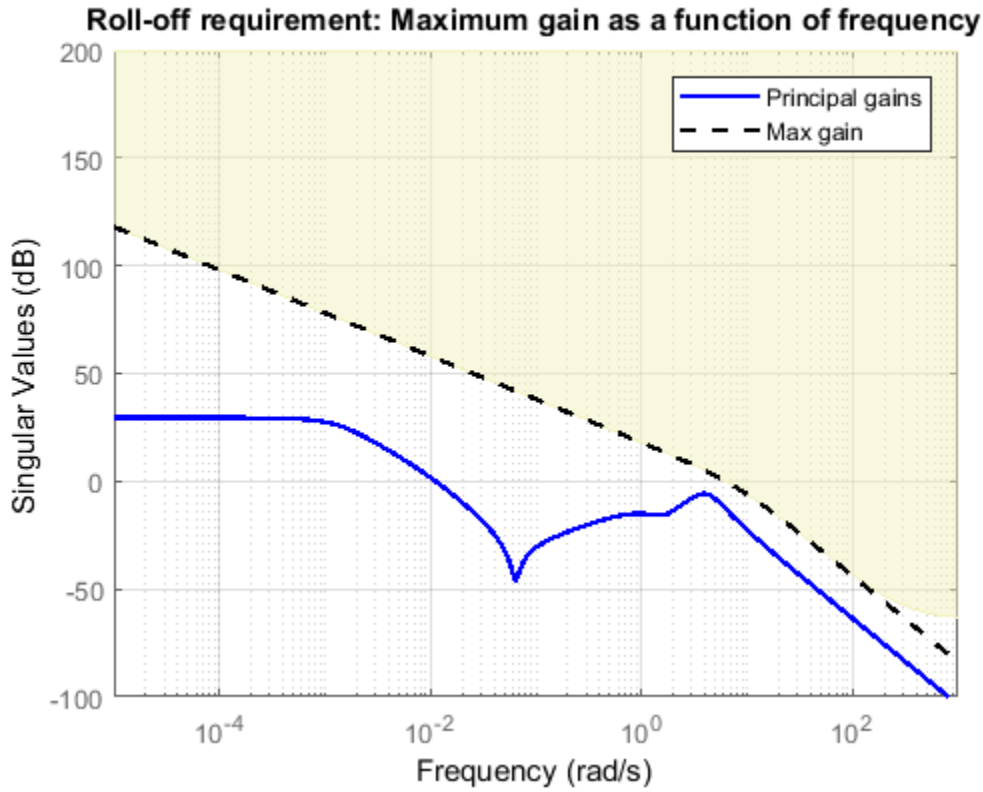
How you obtain tuning-goal plots depends on your work environment.

- At the command line, use `viewGoal`.
- In Control System Tuner, each tuning goal that you create generates a tuning-goal plot. When you tune the control system, these plots update to reflect the tuned design.

The form of the tuning-goal plot depends on the specific tuning goal you use. For instance, for time-domain tuning goals, the tuning-goal plot is a time-domain plot of the relevant system response. The following plot, adapted from the example “MIMO Control of Diesel Engine” (Control System Toolbox), shows a typical tuning-goal plot for a time-domain disturbance-rejection goal. The dashed lines represent the worst acceptable step response specified in the tuning goal. The solid line shows the corresponding response of the tuned system.



Similarly, the plots for frequency-domain tuning goals show the target response and the tuned response in the frequency domain. The following plot, adapted from the example "Fixed-Structure Autopilot for a Passenger Jet" (Control System Toolbox), shows a plot for a gain goal (`TuningGoal.Gain` at the command line). This tuning goal limits the gain between a specified input and output to a frequency-dependent profile. In the plot, the dashed line shows the gain profile specified in the tuning goal. If the tuned system response (solid line) enters the shaded region, the tuning goal is violated. In this case, the tuning goal is satisfied at all frequencies.



Difference Between Dashed Line and Shaded Region

With some frequency-domain tuning goals, there might be a difference between the gain profile you specify in the tuning goal, and the profile the software uses for tuning. In this case, the shaded region of the plot reflects the profile that the software uses for tuning. The gain profile you specify and the gain profile used for tuning might differ if:

- You tune a control system in discrete time, but specify the gain profile in continuous time.
- The software modifies the asymptotes of the specified gain profile to improve numeric stability.

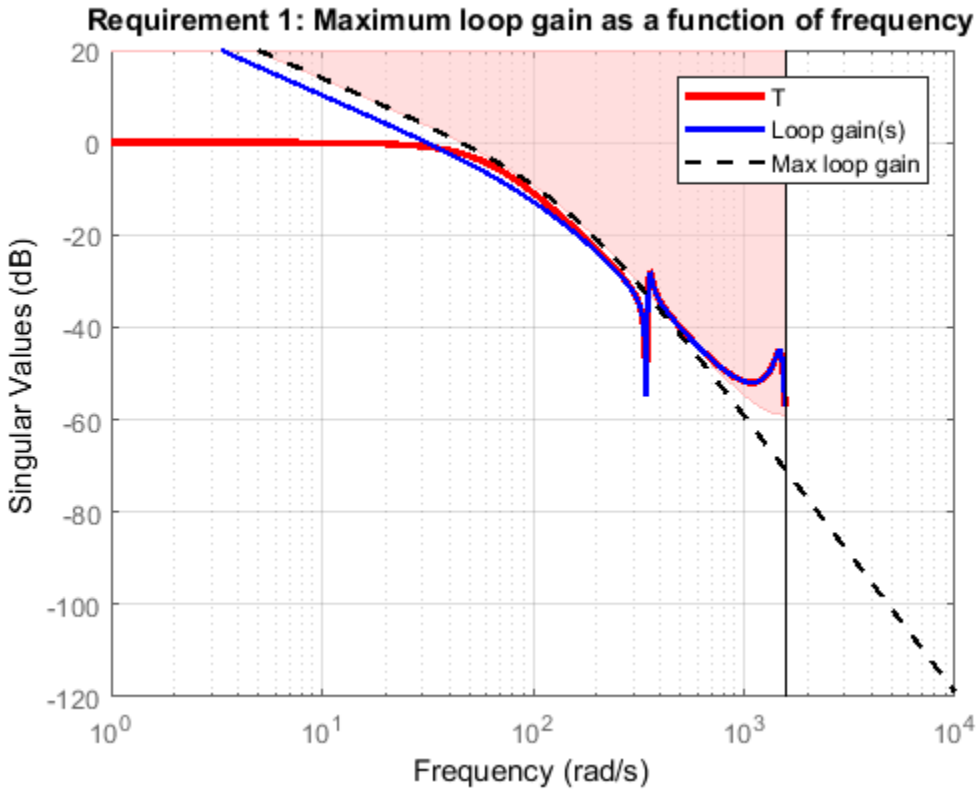
Continuous-Time Gain Profile for Discrete-Time Tuning

When you tune a discrete-time control system, you can specify frequency-dependent tuning goals using discrete-time or continuous-time transfer functions. If you use a continuous-time transfer function, the tuning algorithm discretizes the transfer function before tuning. For instance, suppose that you specify a tuning goal as follows.

```
W = zpk([],[0 -150 -150],1125000);
Req = TuningGoal.MaxLoopGain('Xloc',W);
```

Suppose further that you use the tuning goal with `systune` to tune a discrete-time `genss` model or `slTuner` interface. `CL` is the resulting tuned control system. To examine the result, generate a tuning-goal plot.

```
viewGoal(Req,CL)
```



The plot shows `W`, the continuous-time maximum loop gain that you specified, as a dashed line. The shaded region shows the discretized version of `W` that `systune` uses for tuning. The discretized maximum loop gain cuts off at the Nyquist frequency corresponding to the sample time of `CL`. Near that cutoff, the shaded region diverges from the dashed line.

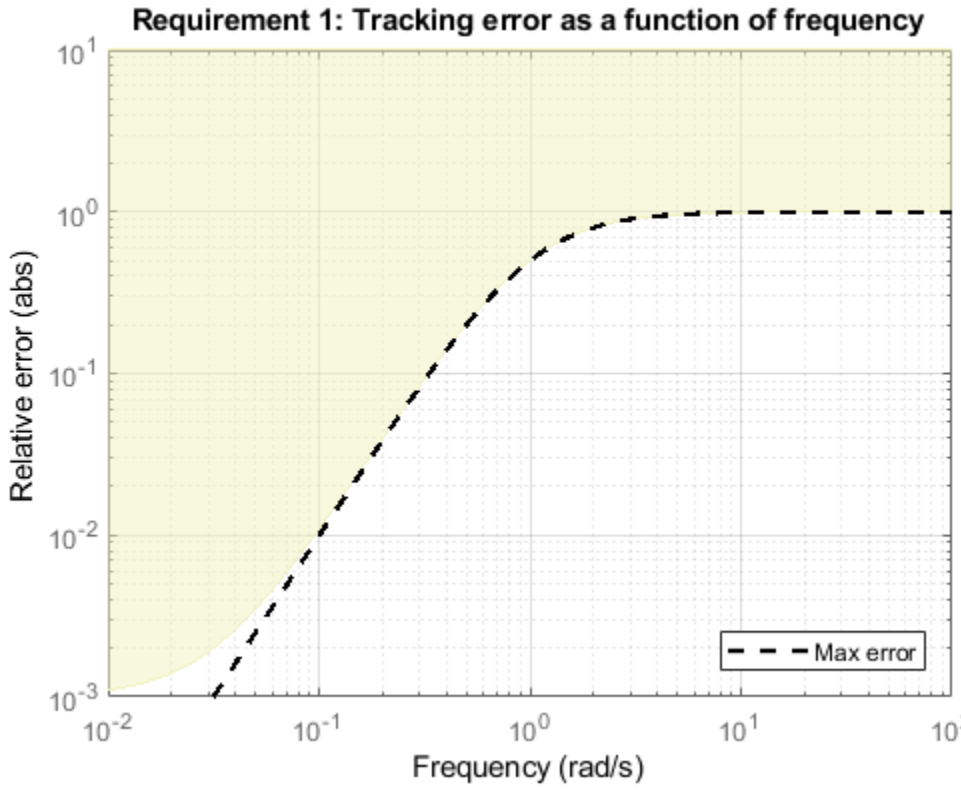
The plot highlights that sometimes it is preferable to specify tuning goals for discrete-time tuning using discrete-time gain profiles. In particular, specifying a discrete-time profile gives you more control over the behavior of the gain profile near the Nyquist frequency.

Modifications for Numeric Stability

When you use a tuning goal with a frequency-dependent specification, the tuning algorithm uses a frequency-weighting function to compute the normalized value of the tuning goal. This weighting function is derived from the gain profile that you specify. For numeric tractability, weighting functions must be stable and proper. For numeric stability, their dynamics must be in the same frequency range as the control system dynamics. For these reasons, the software might adjust the specified gain profile to eliminate undesirable low-frequency or high-frequency dynamics or asymptotes. The process of modifying the tuning goal for better numeric conditioning is called regularization.

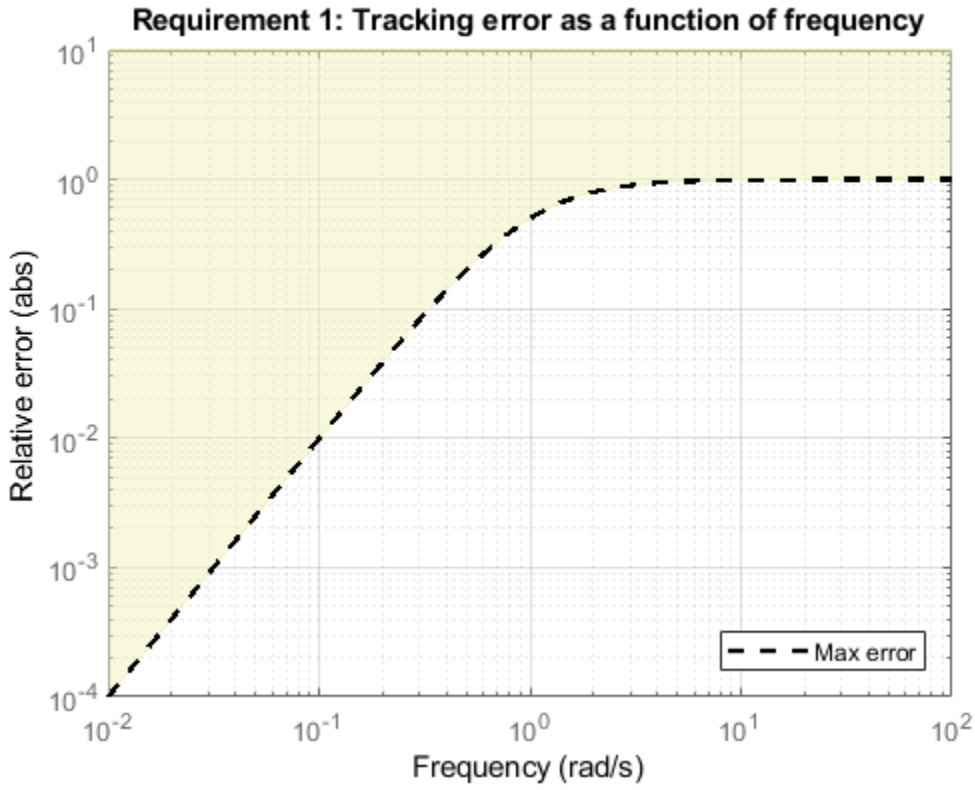
For example, consider the following tracking goal.

```
R1 = TuningGoal.Tracking('r','y',tf([1 0 0],[1 2 1]));
viewGoal(R1)
```



Here the control bandwidth is about 1 rad/s and the gain profile has two zeros at s = 0, which become unstable poles in the weighting function (see `TuningGoal.Tracking` for details). The regularization moves these zeros to about 0.01 rad/s, and the maximum tracking error levels off at about 10^{-3} (0.1%). If you need better tracking accuracy, you can explicitly specify the cutoff frequency in the error profile.

```
R2 = TuningGoal.Tracking('r','y',tf([1 0 5e-8],[1 2 1]));
viewGoal(R2)
set(gca,'Ylim',[1e-4,10])
```



However, for numeric safety, the regularized weighting function always levels off at very low and very high frequencies, regardless of the specified gain profile.

Access the Regularized Functions

When you are working at the command line, you can obtain the regularized gain profile using the `getWeight` or `getWeights` commands. For details, see the reference pages for the individual tuning goals for which the tuning algorithm performs regularization:

- `TuningGoal.Gain`
- `TuningGoal.LoopShape`
- `TuningGoal.MaxLoopGain`

- `TuningGoal.MinLoopGain`
- `TuningGoal.Rejection`
- `TuningGoal.Sensitivity`
- `TuningGoal.StepRejection`
- `TuningGoal.Tracking`

In Control System Tuner, you cannot view the regularized weighting functions directly. Instead, use the tuning-goal commands to generate an equivalent tuning goal, and use `getWeight` or `getWeights` to access the regularized functions.

Improve Tuning Results

If the tuning results do not adequately meet your design requirements, adjust your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced.
  - In Control System Tuner, use the **Enforce goal in frequency range** field of the tuning goal dialog box.
  - At the command line, use the `Focus` property of the `TuningGoal` object.

If the tuning results do satisfy your design requirements, you can validate the tuned control system as described in “Validate Tuned Control System” on page 9-226.

See Also

`viewGoal`

Related Examples

- “Interpret Numeric Tuning Results” on page 9-183
- “Create Response Plots in Control System Tuner” on page 9-197
- “Validate Tuned Control System” on page 9-226

Create Response Plots in Control System Tuner

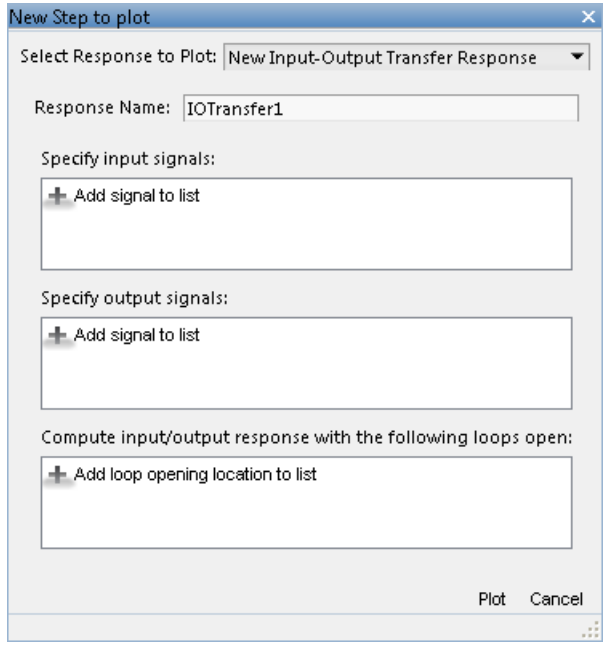
This example shows how to create response plots for analyzing system performance in Control System Tuner. Control System Tuner can generate many types of response plots in the time and frequency domains. You can view responses of SISO or MIMO transfer functions between inputs and outputs at any location in your model. When you tune your control system, Control System Tuner updates the response plots to reflect the tuned design. Use response plots to validate the performance of the tuned control system.

This example creates response plots for analyzing the sample model `rct_helico`.

Choose Response Plot Type

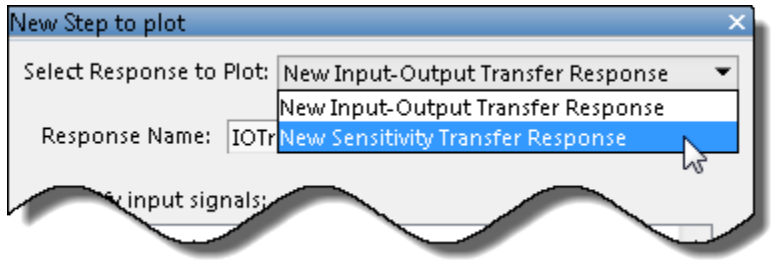
In Control System Tuner, in the **Control System** tab, click **New Plot**. Select the type of plot you want to create.

A new plot dialog box opens in which you specify the inputs and outputs of the portion of your control system whose response you want to plot. For example, select **New step** to create a step response plot from specified inputs to specified outputs of your system.



Specify Transfer Function

Choose which transfer function associated with the specified inputs and outputs you want to analyze.



For most response plots types, the **Select Response to Plot** menu lets you choose one of the following transfer functions:

- `New Input-Output Transfer Response` — Transfer function between specified inputs and outputs, computed with loops open at any additionally specified loop-opening locations.
- `New Sensitivity Transfer Response` — Sensitivity function computed at the specified location and with loops open at any specified loop-opening locations.
- `New Open-Loop Response` — Open loop point-to-point transfer function computed at the specified location and with loops open at any additionally specified loop-opening locations.
- `Entire System Response` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations for the entire closed-loop control system.
- `Response of Tuned Block` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations of tuned blocks.

Name the Response

Type a name for the response in the **Response Name** text box. Once you have specified signal locations defining the response, Control System Tuner stores the response under this name. When you create additional new response plots, the response appears by this name in **Select Response to Plot** menu.

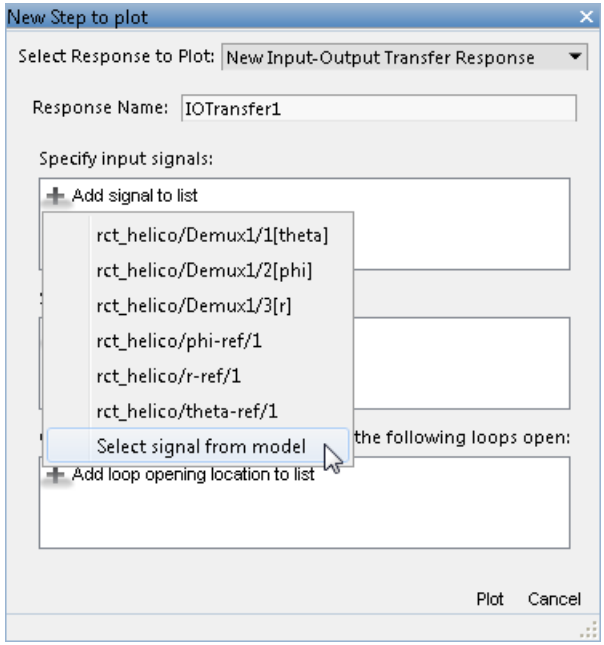
Choose Signal Locations for Evaluating System Response

Specify the signal locations in your control system at which to evaluate the selected response. For example, the step response plot displays the response of the system at one

or more output locations to a unit step applied at one or more input locations. Use the **Specify input signals** and **Specify output signals** sections of the dialog box to specify these locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

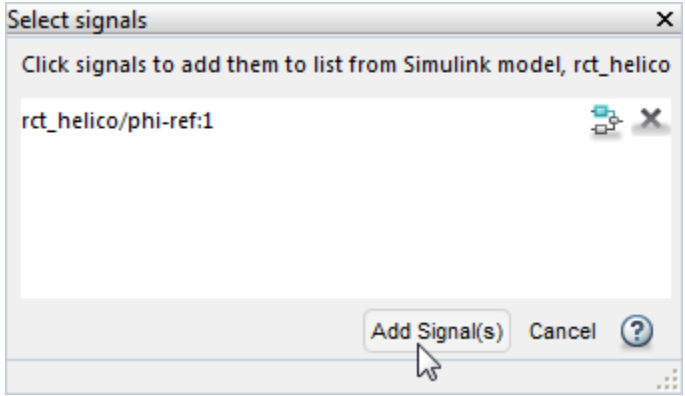
Under **Specify input signals**, click **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.



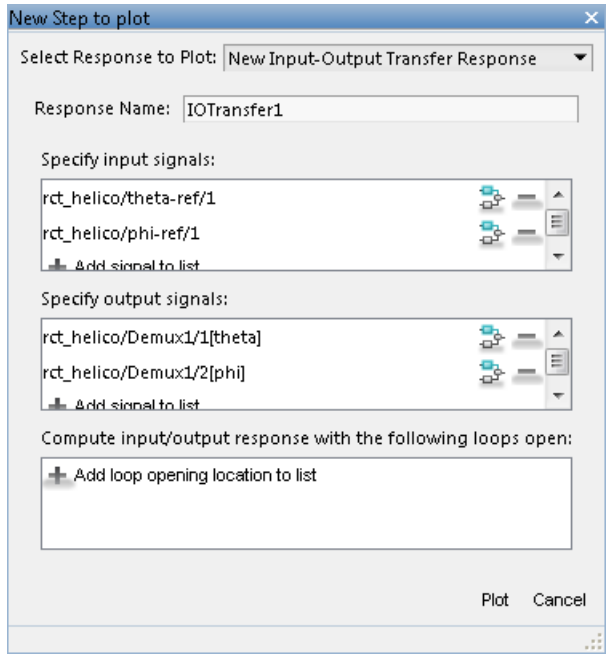
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO response, and click multiple signals to create a MIMO response.

Click **Add signal(s)**. The **Select signals** dialog box closes.



The signal or signals you selected now appear in the list of step-response inputs in the new-plot dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration plots the MIMO response to a step input applied at `theta-ref` and `phi-ref` and measured at `theta` and `phi` in the Simulink model `rct_helico`.



Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and .

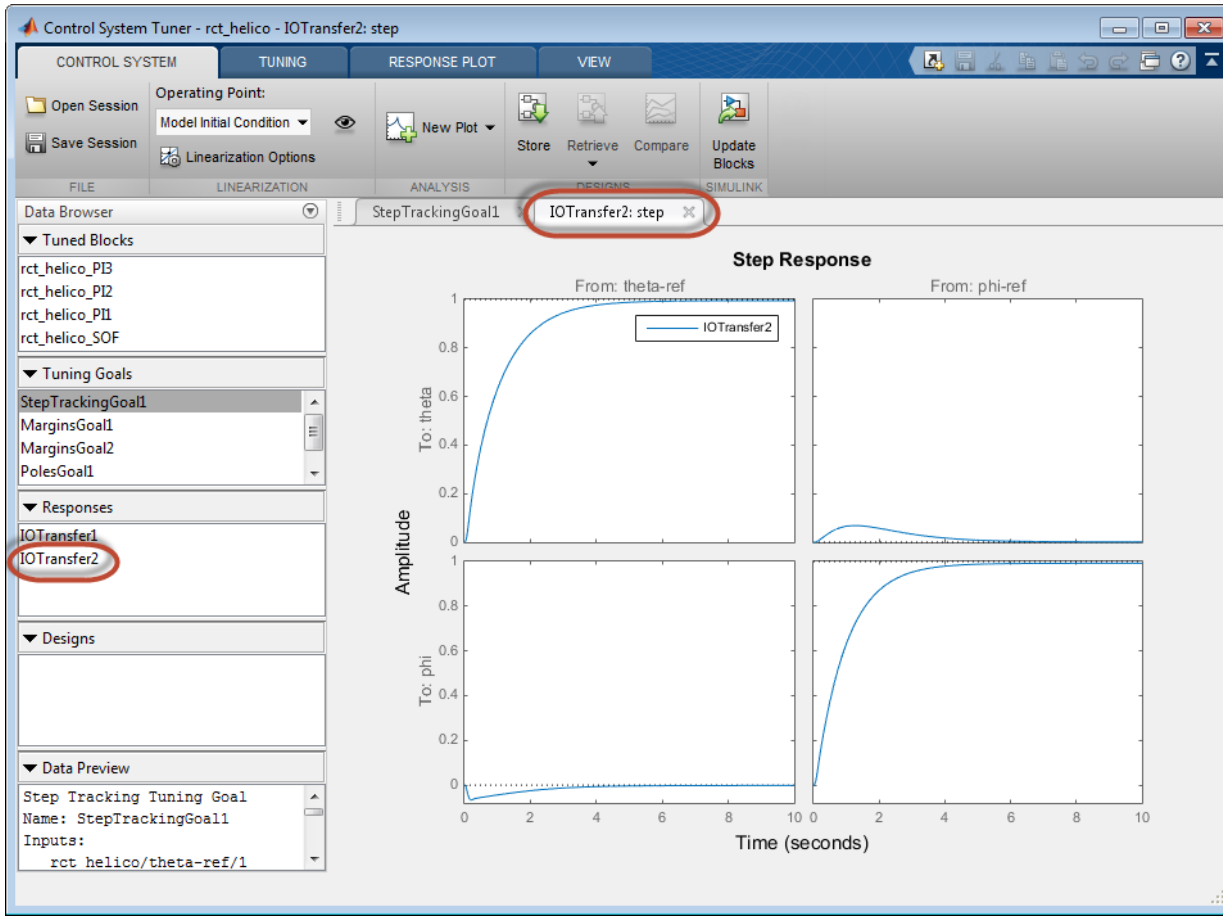
Specify Loop Openings

You can evaluate most system responses with loops open at one or more locations in the control system. Click **Add loop opening location to list** to specify such locations for the response.

Store and Plot the Response

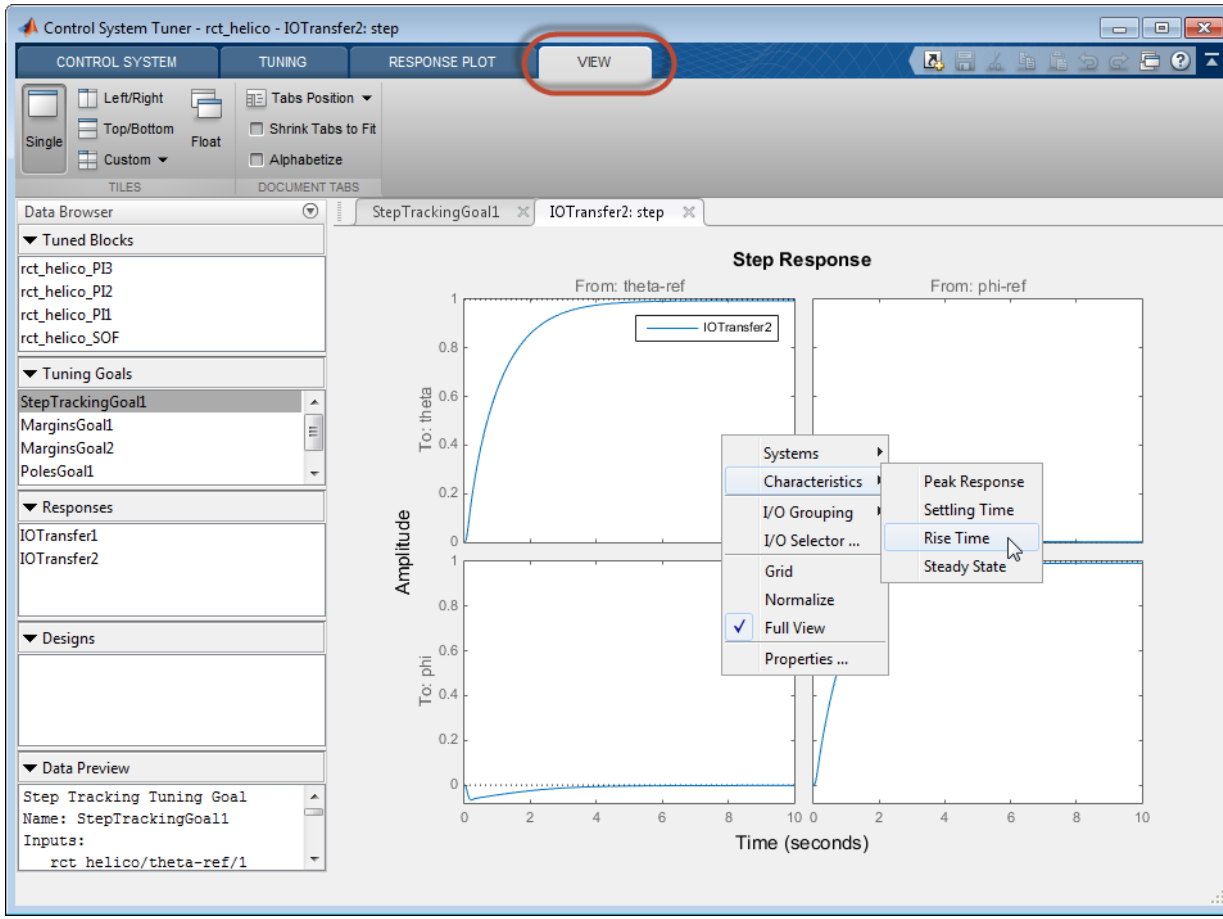
When you have finished specifying the response, click **Plot** in the new plot dialog box. The new response appears in the **Responses** section of the Data Browser. A new figure

opens displaying the response plot. When you tune your control system, you can refer to this figure to evaluate the performance of the tuned system.



Tip To edit the specifications of the response, double-click the response in the Data Browser. Any plots using that response update to reflect the edited response.

View response characteristics such as rise-times or peak values by right-clicking on the plot. Other options for managing and organizing multiple plots are available in the **View** tab.



See Also

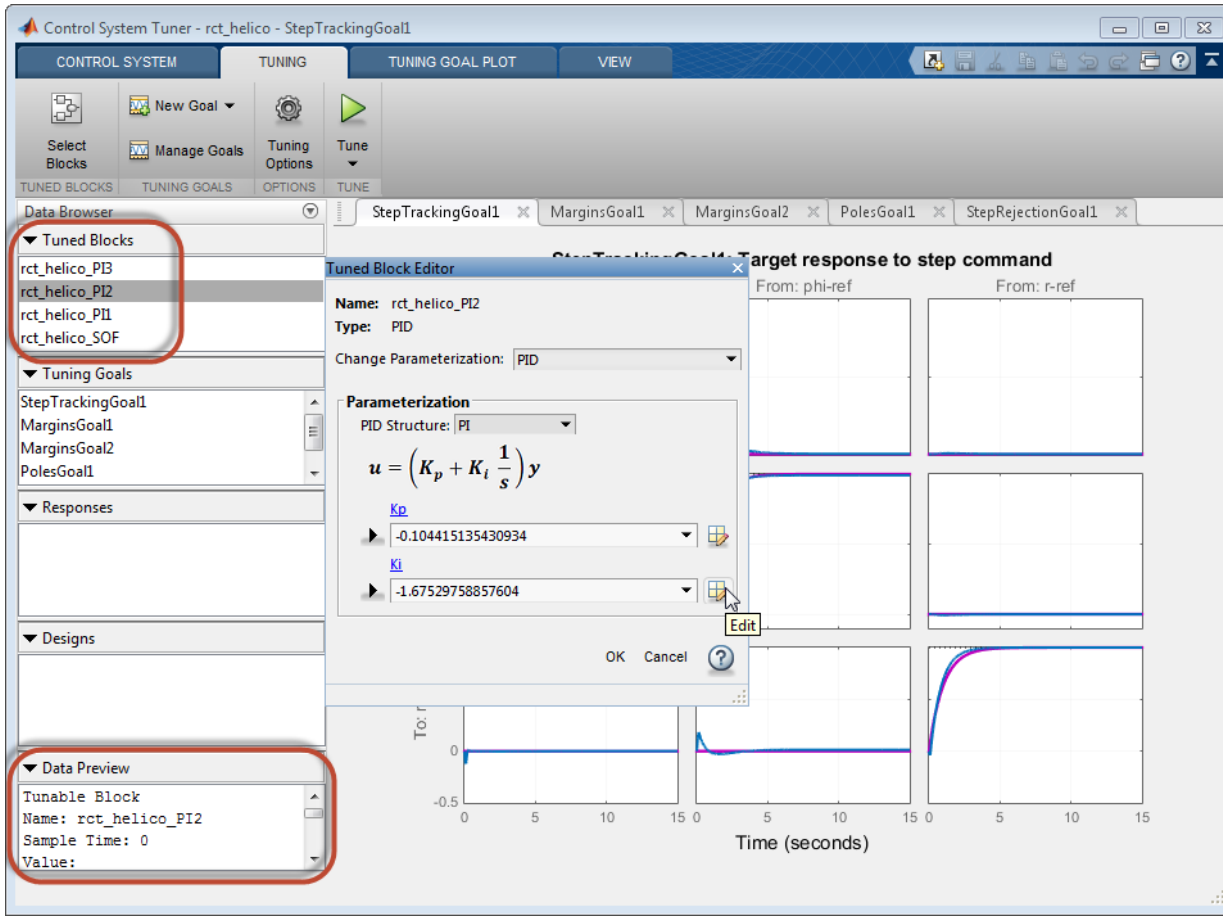
Related Examples

- “Compare Performance of Multiple Tuned Controllers” on page 9-206
- “Examine Tuned Controller Parameters in Control System Tuner” on page 9-204
- “Visualize Tuning Goals” on page 9-188

Examine Tuned Controller Parameters in Control System Tuner

After you tune your control system, Control System Tuner gives you two ways to view the current values of the tuned block parameters:

- In the Data Browser, in the **Tuned Blocks** area, select the block whose parameters you want to view. A text summary of the block and its current parameter values appears in the Data Browser in the **Data Preview** area.
- In the Data Browser, in the **Tuned Blocks** area, double-click the block whose parameters you want to view. The Tuned Block Editor opens, displaying the current values of the parameters. For array-valued parameters, click  to open a variable editor displaying values in the array.



Note To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

See Also

Related Examples

- “View and Change Block Parameterization in Control System Tuner” on page 9-27

Compare Performance of Multiple Tuned Controllers

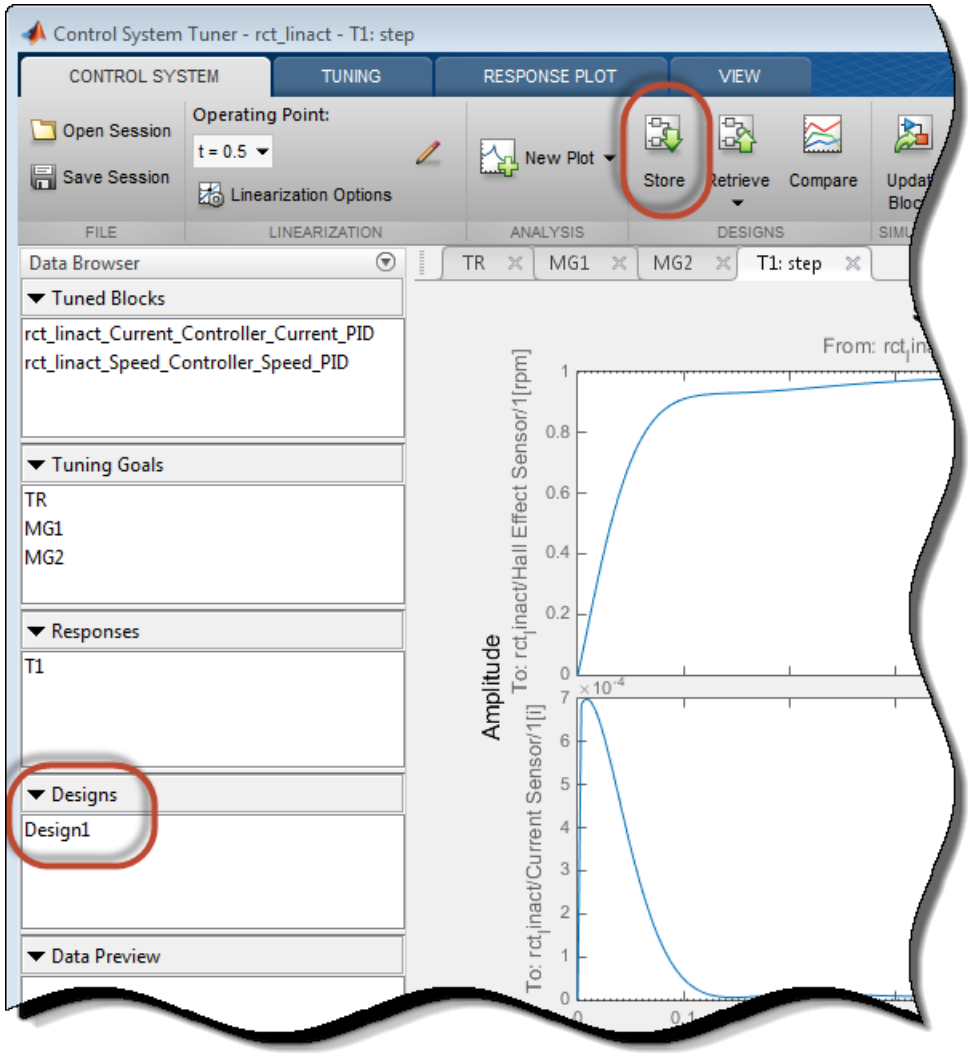
Control System Tuner lets you compare the performance of a control system tuned with two different sets of tuning goals. Such comparison is useful, for example, to see the effect on performance of changing a tuning goal from hard goal to soft goal. Comparing performance is also useful to see the effect of adding an additional tuning goal when an initial design fails to satisfy all your performance requirements either in the linearized system or when validated against a full nonlinear model.

This example compares tuning results for the sample model `rct_linact`.

Store First Design

After tuning a control system with a first set of design requirements, store the design in Control System Tuner.

In the **Control System** tab, click  **Store**. The stored design appears in the Data Browser in the **Designs** area.



Change the name of the stored design, if desired, by right-clicking on the data browser entry.

Compute New Design

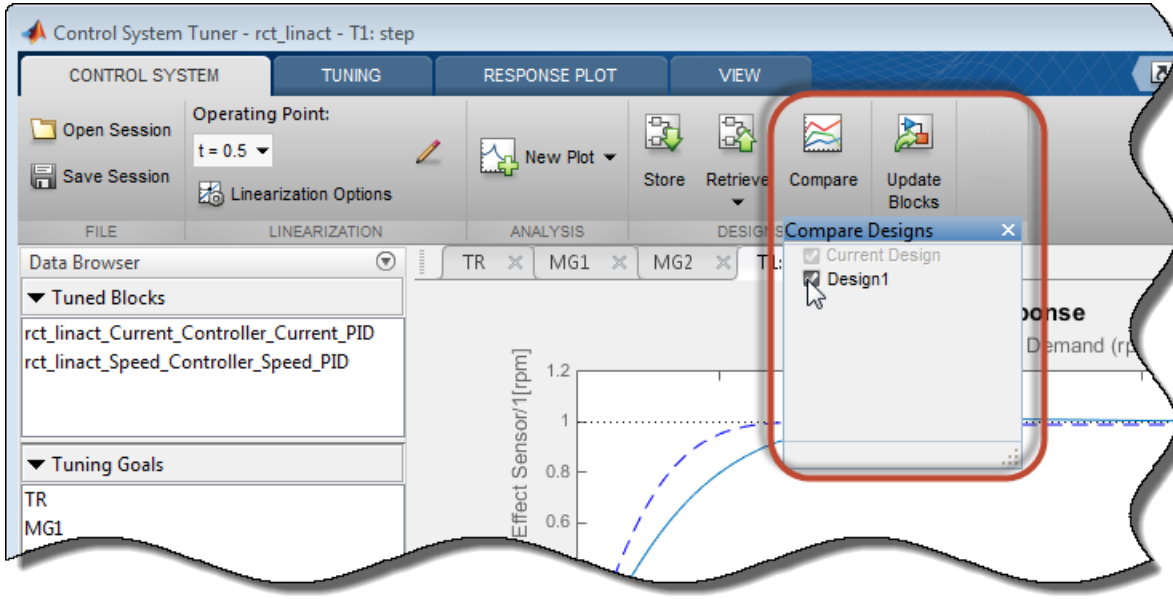
In the **Tuning** tab, make any desired changes to the tuning goals for the second design. For example, add new tuning goals or edit existing tuning goals to change specifications.

Or, in **Manage Goals**, change which tuning goals are active and which are designated hard constraints or soft requirements.

When you are ready, retune the control system with the new set of tuning goals. Click **Tune**. Control System Tuner updates the current design (the current set of controller parameters) with the new tuned design. All existing plots, which by default show the current design, are updated to reflect the new current design.

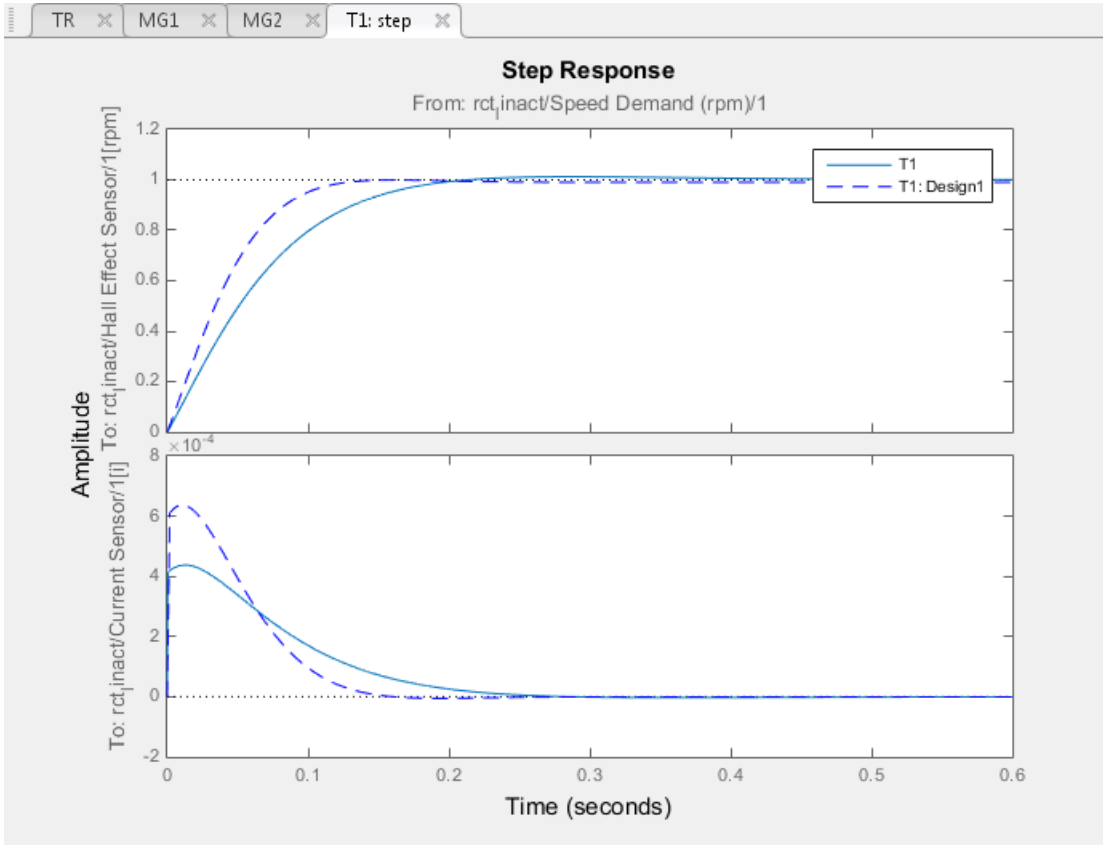
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design. In the **Control System** tab, click **Compare**. The **Compare Designs** dialog box opens.



In the **Compare Designs** dialog box, the current design is checked by default. Check the box for the design you want to compare to the current design. All response plots and

tuning goal plots update to reflect the checked designs. The solid trace corresponds to the current design. Other designs are identified by name in the plot legend.

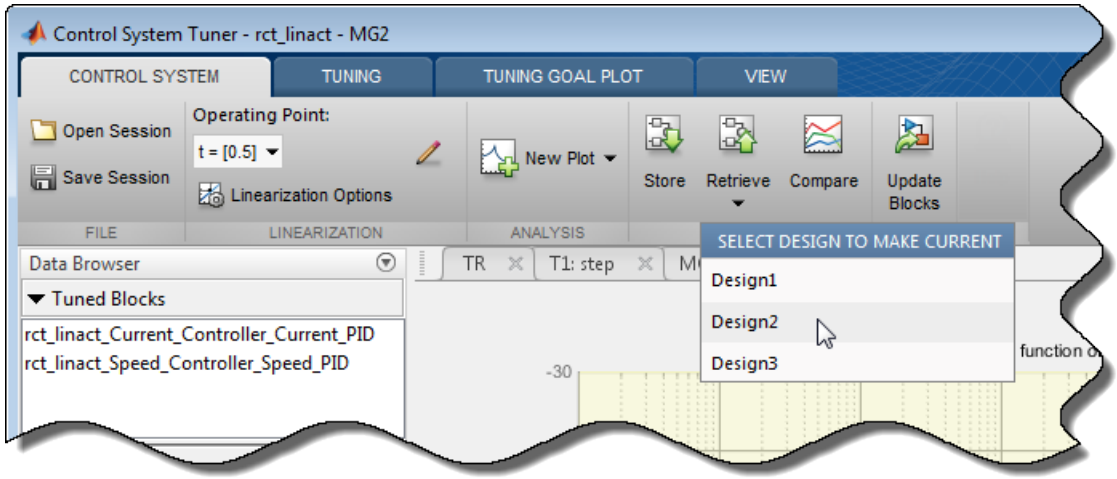


Use the same procedure save and compare as many designs as you need.

Restore Previously Saved Design

Under some conditions, it is useful to restore the tuned parameter values from a previously saved design as the current design. For example, clicking **Update Blocks** writes the current parameter values to the Simulink model. If you decide to test a stored controller design in your full nonlinear model, you must first restore those stored values as the current design.

To do so, click **Retrieve**. Select the stored design that you want to make the current design.



See Also

Related Examples

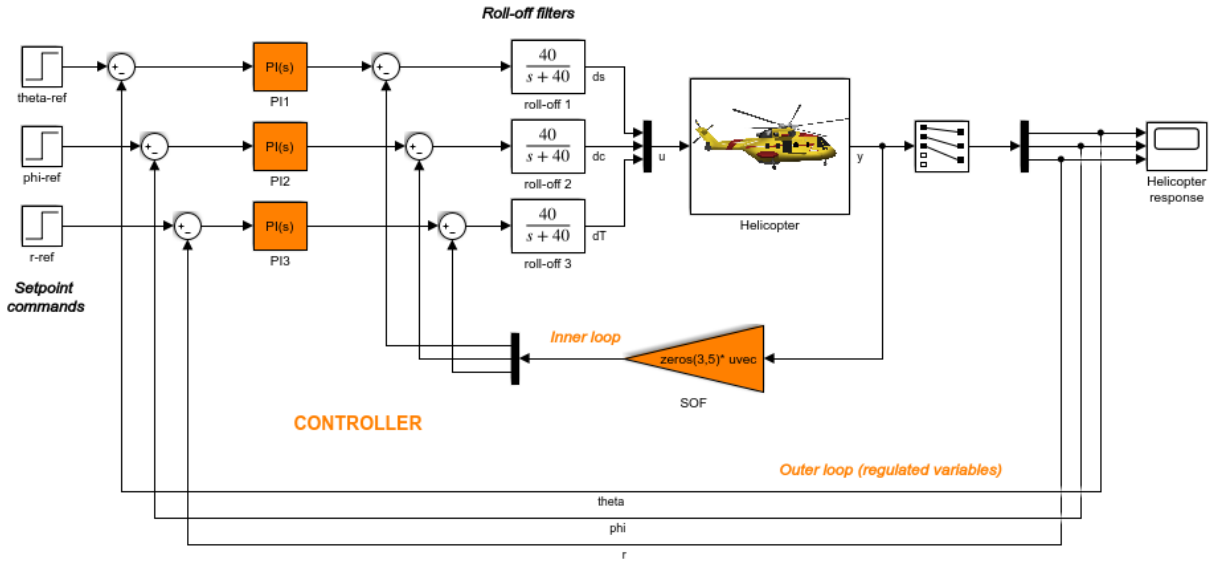
- "Create Response Plots in Control System Tuner" on page 9-197

Create and Configure slTuner Interface to Simulink Model

This example shows how to create and configure an `slTuner` interface for a Simulink® model. The `slTuner` interface parameterizes blocks in your model that you designate as tunable and allows you to tune them using `systune`. The `slTuner` interface generates a linearization of your Simulink model, and also allows you to extract linearized system responses for analysis and validation of the tuned control system.

For this example, create and configure an `slTuner` interface for tuning the Simulink model `rct_helico`, a multiloop controller for a rotorcraft. Open the model.

```
open_system('rct_helico');
```



The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance.

Suppose that you want to tune this model to meet the following control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, specified rise times, minimal overshoot, and minimal cross-coupling.

- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise.
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs).

The `systune` command can jointly tune the controller blocks `SOF` and the PI controllers to meet these design requirements. The `slTuner` interface sets up this tuning task.

Create the `slTuner` interface.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

This command initializes the `slTuner` interface with the three PI controllers and the `SOF` block designated as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. First, add the outputs and reference inputs for the tracking requirements.

```
addPoint(ST0,{'theta-ref','theta','phi-ref','phi','r-ref','r'});
```

When you create a `TuningGoal.Tracking` object that captures the tracking requirement, this object references the same signals.

Configure the `slTuner` interface for the stability margin requirements. Designate as analysis points the plant inputs and outputs (control and measurement signals) where the stability margins are measured.

```
addPoint(ST0,{'u','y'});
```

Display a summary of the `slTuner` interface configuration in the command window.

```
ST0
```

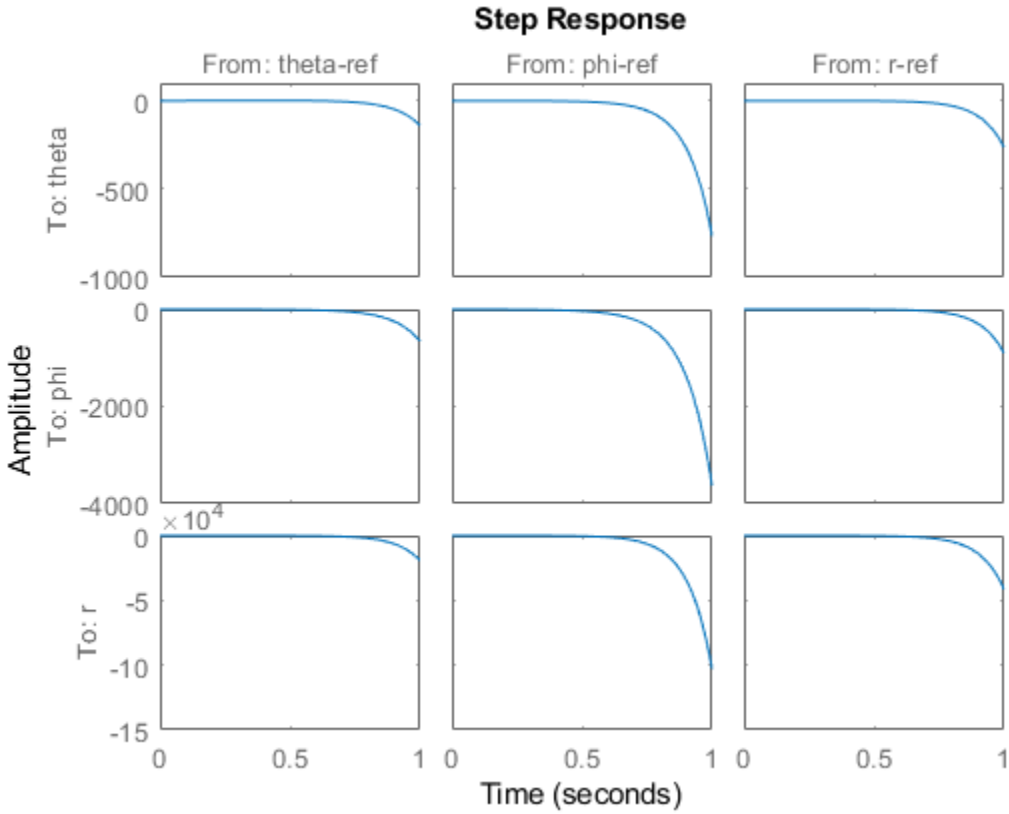
```
slTuner tuning interface for "rct_helico":

4 Tuned blocks: (Read-only TunedBlocks property)
--------------------------
Block 1: rct_helico/PI1
Block 2: rct_helico/PI2
Block 3: rct_helico/PI3
```

```
Block 4: rct_helico/SOF

8 Analysis points:
--------------------------
Point 1: Port 1 of rct_helico/theta-ref
Point 2: Signal "theta", located at port 1 of rct_helico/Demux1
Point 3: Port 1 of rct_helico/phi-ref
Point 4: Signal "phi", located at port 2 of rct_helico/Demux1
Point 5: Port 1 of rct_helico/r-ref
Point 6: Signal "r", located at port 3 of rct_helico/Demux1
Point 7: Signal "u", located at port 1 of rct_helico/Mux3
Point 8: Signal "y", located at port 1 of rct_helico/Helicopter

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

In the command window, click on any highlighted signal to see its location in the Simulink model.

In addition to specifying design requirements, you can use analysis points for extracting system responses. For example, extract and plot the step responses between the reference signals and `'theta'`, `'phi'`, and `'r'`.

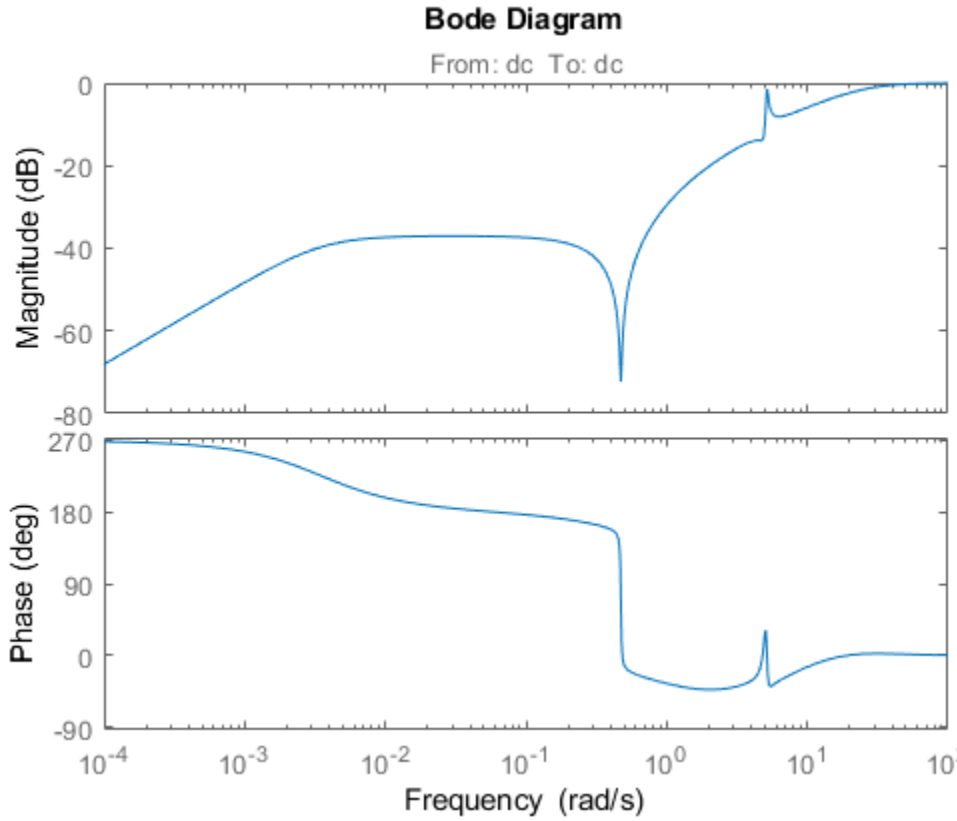
```
T0 = getIOTransfer(ST0,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
stepplot(T0,1)
```



All the step responses are unstable, including the cross-couplings, because this model has not yet been tuned.

After you tune the model, you can similarly use the designated analysis points to extract system responses for validating the tuned system. If you want to examine system responses at locations that are not needed to specify design requirements, add these locations to the `slTuner` interface as well. For example, plot the sensitivity function measured at the output of the block `roll-off 2`.

```
addPoint(ST0,'dc')
dcS0 = getSensitivity(ST0,'dc');
bodeplot(dcS0)
```



Suppose you want to change the parameterization of tunable blocks in the `slTuner` interface. For example, suppose that after tuning the model, you want to test whether changing from PI to PID controllers yields improved results. Change the parameterization of the three PI controllers to PID controllers.

```
PID0 = pid(0,0.001,0.001,.01);  % initial value for PID controllers
PID1 = tunablePID('C1',PID0);
PID2 = tunablePID('C2',PID0);
PID3 = tunablePID('C3',PID0);

setBlockParam(ST0,'PI1',PID1,'PI2',PID2,'PI3',PID3);
```

After you configure the `slTuner` interface to your Simulink model, you can create tuning goals and tune the model using `systune` or `looptune`.

See Also

`addBlock` | `addPoint` | `getIOTransfer` | `getSensitivity` | `setBlockParam` | `slTuner`

Related Examples

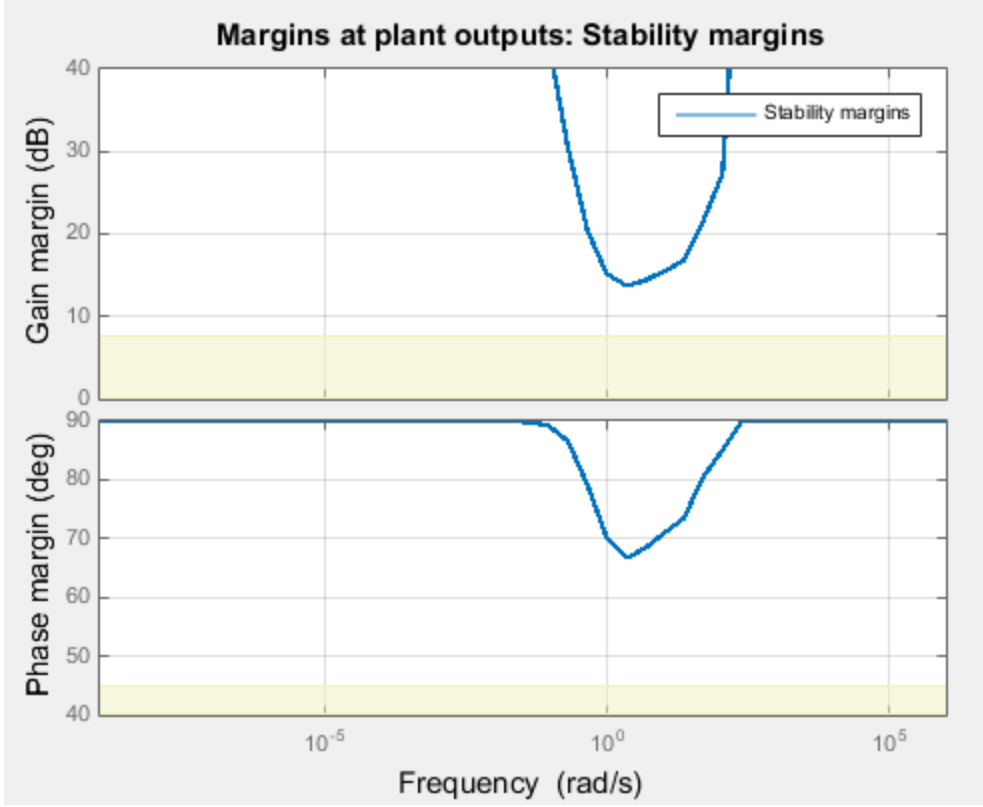
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-48
- “Multiloop Control of a Helicopter” (Control System Toolbox)
- “Control of a Linear Electric Actuator” (Control System Toolbox)

Stability Margins in Control System Tuning

Control System Tuner and `viewGoal` display stability margins as a function of frequency.

Stability Margins Plot

The following plot shows a typical result of tuning a control system with `systune` or Control System Tuner when you use a tuning goal that constrains stability margins.



. You obtain this plot in one of the following ways:

- Tuning in Control System Tuner using a “Margins Goal” on page 9-147 or “Quick Loop Tuning” on page 9-59.

- Tuning at the command line using `systune` with `TuningGoal.Margins`. If `S` is the control system model or `slTuner` interface, and `Req` is a `TuningGoal.Margins` goal, obtain the stability-margin plot by entering:

  ```
  viewGoal(S,Req)
  ```

Gain and Phase Margins

For SISO systems, the gain and phase margins at a frequency ω indicate how much the gain or phase of the open-loop response $L(j\omega)$ can change without loss of stability. For example, a gain margin of 5dB at 2 rad/s indicates that closed-loop stability is maintained when the loop gain increases or decreases by as much as 5dB at this frequency. Gain and phase margins typically vary across frequencies.

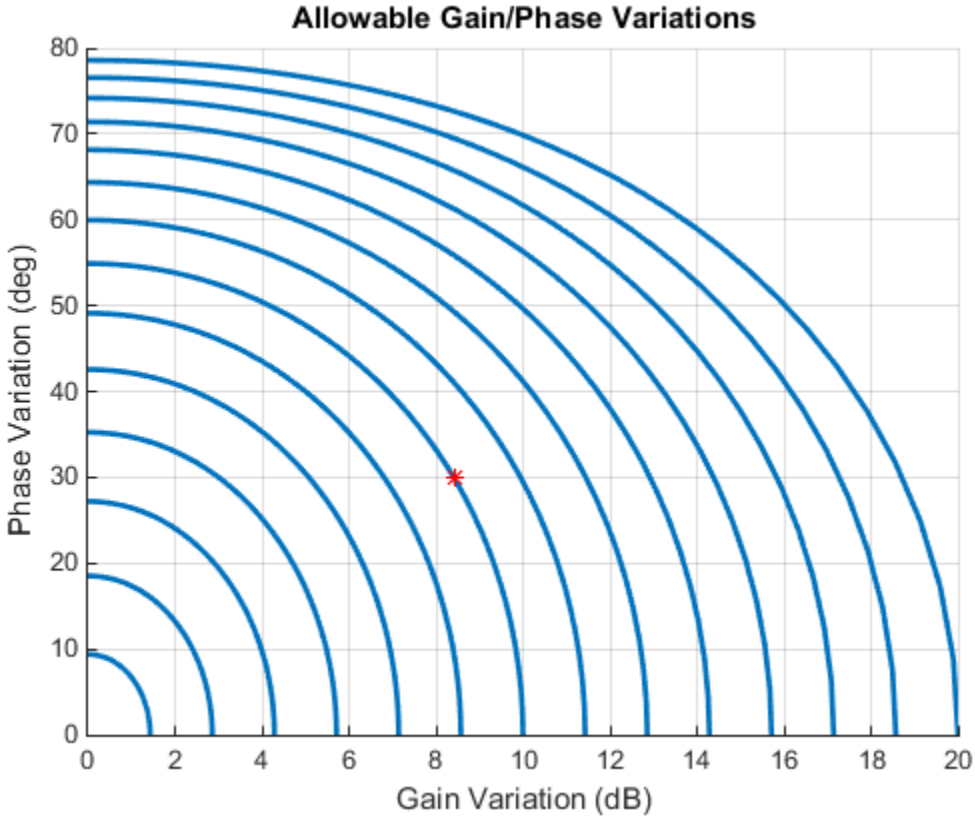
For MIMO systems, gain and phase margins are interpreted as follows:

- Gain margin: Stability is preserved when the gain increases or decreases by up to the gain margin value in each channel of the feedback loop.
- Phase margin: Stability is preserved when the phase increases or decreases by up to the phase margin value in each channel of the feedback loop.

In MIMO systems, the gain or phase can change in all channels at once, and by a different amount in each channel. The Margins Goal and `TuningGoal.Margins` rely on the notion of disk margin for MIMO systems. (See “Algorithm” on page 9-221.) Like SISO stability margins, gain and phase margins in MIMO systems typically vary across frequency.

Combined Gain and Phase Variations

To assess robustness to changes in both gain and phase, use the following chart.

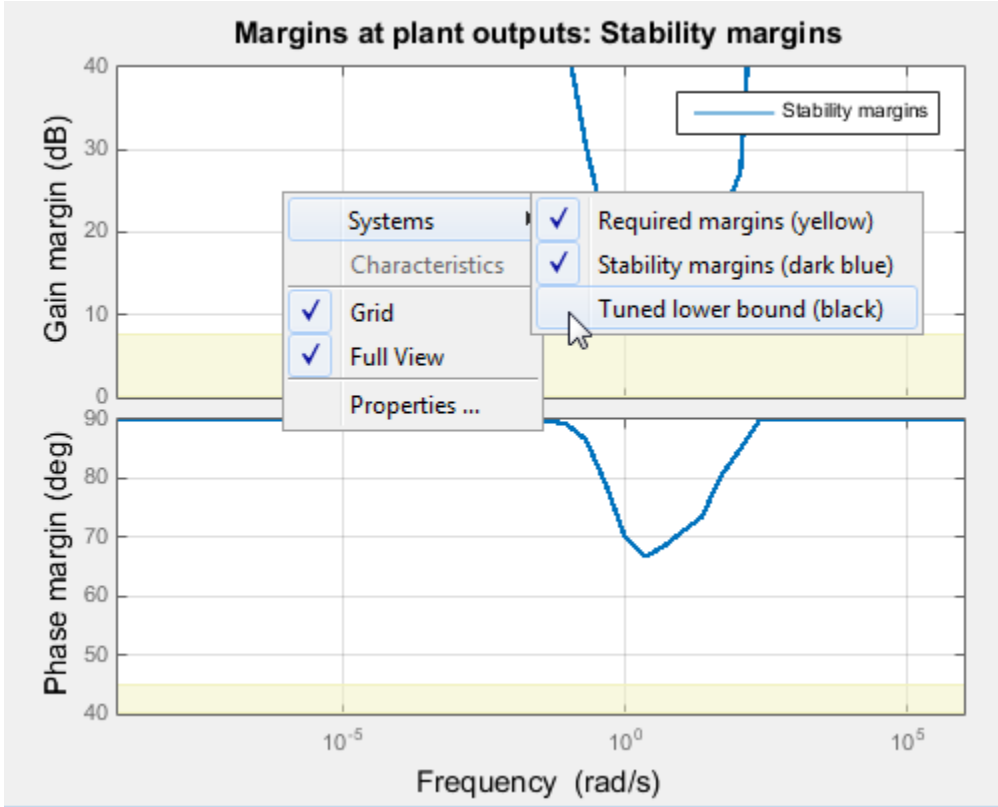


For example, if the gain margin plot in Control System Tuner indicates a 10 dB margin at a particular frequency, then trace the contour starting at `(Gain,Phase) = (10,0)` to see how a given amount of phase variation reduces the allowable gain variation at that frequency. For example, if the phase can vary by 30 degrees than the gain can only vary by about 8.4 dB (red mark).

Interpreting the Gain and Phase Margin Plot

The stability-margin plot for Margins Goal or `TuningGoal.Margins` shows in shaded yellow the region where the target margins are not met. The plot displays the current gain and phase margins (computed using the current values of the tunable parameters in the control system) as a blue trace.

These gain and phase margin curves are obtained using an exact calculation involving μ-analysis. For computational efficiency, however, the tuning algorithm uses an approximate calculation that can yield smaller margins in parts of the frequency range. To see the lower bound used by the tuner, right-click on the plot, and select **Systems** > **Tuned Lower Bound**.



If there is a significant gap between the true margins and the tuner approximation, try increasing the D-scaling order. The default order is zero (static scaling). For tuning in Control System Tuner, set the D-scaling order in the Margins Goal dialog box. For command-line tuning, set this value using the `ScalingOrder` property of `TuningGoal.Margins`.

Algorithm

The gain and phase margin values are both derived from the disk margin (see `loopmargin`). The disk margin measures the radius of a circular exclusion region centered near the critical point. This radius is a decreasing function of the scaled norm:

$$\min_{D\ \text{diagonal}} \left\| D^{-1}\left(I - L(j\omega)\right)\left(I + L(j\omega)\right)^{-1} D \right\|_2 .$$

Unlike the traditional gain and phase margins, the disk margins and associated gain and phase margins guarantee that the open-loop response $L(j\omega)$ stays at a safe distance from the critical point at all frequencies.

See Also

`TuningGoal.Margins` | `loopmargin`

More About

- "Loop Shape and Stability Margin Specifications" (Control System Toolbox)
- "Margins Goal" on page 9-147

Tune Control System at the Command Line

After specifying your tuning goals using `TuningGoal` objects (see “Tuning Goals”), use `systune` to tune the parameters of your model.

The `systune` command lets you designate one or more design goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have tuning goals.`systune` attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

Organize your `TuningGoal` objects into a vector of soft requirements and a vector of hard requirements. For example, suppose you have created a tracking requirement, a rejection requirement, and stability margin requirements at the plant inputs and outputs. The following commands tune the control system represented by `T0`, treating the stability margins as hard goals, the tracking and rejection requirements as soft goals. (`T0` is either a `genss` model or an `slTuner` interface previously configured for tuning.)

```
SoftReqs = [Rtrack,Rreject];
HardReqs = [RmargIn,RmargOut];
[T,fSoft,gHard] = systune(T0,SoftReqs,HardReqs);
```

`systune` converts each tuning requirement into a normalized scalar value, f for the soft constraints and g for the hard constraints. The command adjusts the tunable parameters of `T0` to minimize the f values, subject to the constraint that each $g < 1$. `systune` returns the vectors `fSoft` and `gHard` that contain the final normalized values for each tuning goal in `SoftReqs` and `HardReqs`.

Use `systuneOptions` to configure additional options for the `systune` algorithm, such as the number of independent optimization runs, convergence tolerance, and output display options.

See Also

`systune` | `systune (for slTuner)` | `systuneOptions`

More About

- “Interpret Numeric Tuning Results” on page 9-183

Speed Up Tuning with Parallel Computing Toolbox Software

If you have the Parallel Computing Toolbox software installed, you can speed up the tuning of fixed-structure control systems. When you run multiple randomized optimization starts with `systune`, `looptune`, or `hinfstruct`, parallel computing speeds up tuning by distributing the optimization runs among workers.

To distribute randomized optimization runs among workers:

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences (Parallel Computing Toolbox), manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `systuneOptions`, `looptuneOptions`, or `hinfstructOptions` set that specifies multiple random starts. For example, the following options set specifies 20 random restarts to run in parallel for tuning with `looptune`:

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the options set when you call the tuning command. For example, if you have already created a tunable control system model, `CL0`, and tunable controller, and tuning requirement vectors `SoftReqs` and `HardReqs`, the following command uses parallel computing to tune the control system of `CL0` with `systune`.

```
[CL,fSoft,gHard,info] = systune(CL0,SoftReq,Hardreq,options);
```

To learn more about configuring a parallel pool, see the Parallel Computing Toolbox documentation.

See Also

`parpool`

Related Examples

- “Using Parallel Computing to Accelerate Tuning” (Control System Toolbox)

More About

- “Specify Your Parallel Preferences” (Parallel Computing Toolbox)

Validate Tuned Control System

When you tune a control system using `systune` or Control System Tuner, you must validate the results of tuning. The tuning results provide numeric and graphical indications of how well your tuning goals are satisfied. (See “Interpret Numeric Tuning Results” on page 9-183 and “Visualize Tuning Goals” on page 9-188.) Often, you want to examine other system responses using the tuned controller parameters. If you are tuning a Simulink model, you must also validate the tuned controller against the full nonlinear system. At the command line and in Control System Tuner, there are several tools to help you validate the tuned control system.

Extract and Plot System Responses

In addition to the system responses corresponding to your tuning goals (see “Visualize Tuning Goals” on page 9-188), you can evaluate the tuned system performance by plotting other system responses. For instance, evaluate reference tracking or overshoot performance by plotting the step response of transfer function from the reference input to the controlled output. Or, evaluate stability margins by examining an open-loop transfer function. You can extract any transfer function you need for analysis from the tuned model of your control system.

Extract System Responses at the Command Line

The tuning tools include analysis functions that let you extract responses from your tuned control system.

For generalized state-space (`genss`) models, use:

- `getIOTransfer`
- `getLoopTransfer`
- `getSensitivity`
- `getCompSensitivity`

For an `slTuner` interface, use:

- `getIOTransfer (for slTuner)`
- `getLoopTransfer (for slTuner)`
- `getSensitivity (for slTuner)`

- `getCompSensitivity (for slTuner)`

In either case, the extracted responses are represented by state-space (`ss`) models. You can analyze these models using commands such as `step`, `bode`, `sigma`, or `margin`.

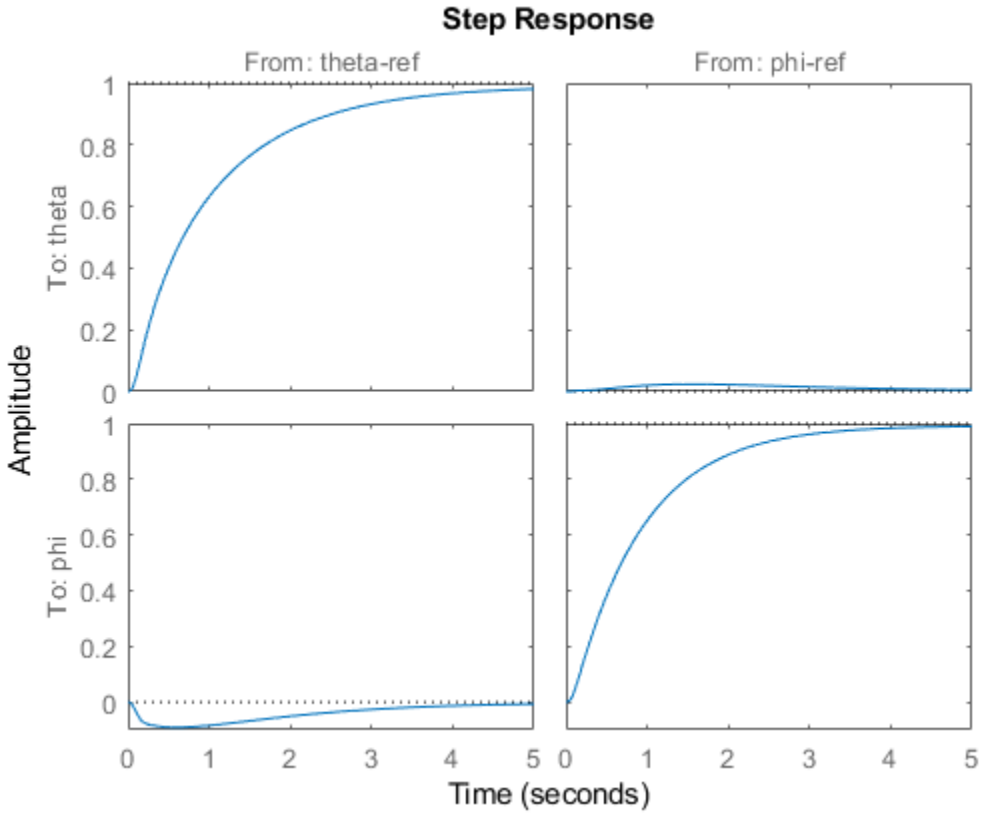
For instance, suppose that you are tuning the control system of the example "Multiloop Control of a Helicopter" (Control System Toolbox). You have created an `slTuner` interface `ST0` for the Simulink model. You have also specified tuning goals `TrackReq`, `MarginReq1`, `MarginReq2`, and `PoleReq`. You tune the control system using `systune`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];
ST1 = systune(ST0,AllReqs);
```

```
Final: Soft = 1.11, Hard = -Inf, Iterations = 100
```

Suppose also that `ST0` has analysis points that include signals named `theta-ref`, `theta`, `phi-ref`, and `phi`. Use `getIOTransfer` to extract the tuned transfer functions from `theta-ref` and `phi-ref` to `theta` and `phi`.

```
T1 = getIOTransfer(ST1,{'theta-ref','phi-ref'},{'theta','phi'});
step(T1,5)
```



The step plot shows that the extracted transfer function is the 2-input, 2-output response from the specified reference inputs to the specified outputs.

For an example that shows how to extract responses from a tuned `genss` model, see "Extract Responses from Tuned MATLAB Model at the Command Line" on page 9-231.

For additional examples, see "Validating Results" (Control System Toolbox).

System Responses in Control System Tuner

For information about extracting and plotting system responses in Control System Tuner, see "Create Response Plots in Control System Tuner" on page 9-197.

Validate Design in Simulink Model

When you tune a Simulink model, the software evaluates tuning goals for a linearization of the model. Similarly, analysis commands such as `getIOTransfer` extract linearized system responses. Therefore, you must validate the tuned controller parameters by simulating the full nonlinear model with the tuned controller parameters, even if the tuned linear system meets all your design requirements. To do so, write the tuned parameter values to the model.

Tip If you tune the Simulink model at an operating point other than the model initial condition, initialize the model at the same operating point before validating the tuned controller parameters. See "Simulate Simulink Model at Specific Operating Point" on page 1-83.

Write Parameters at the Command Line

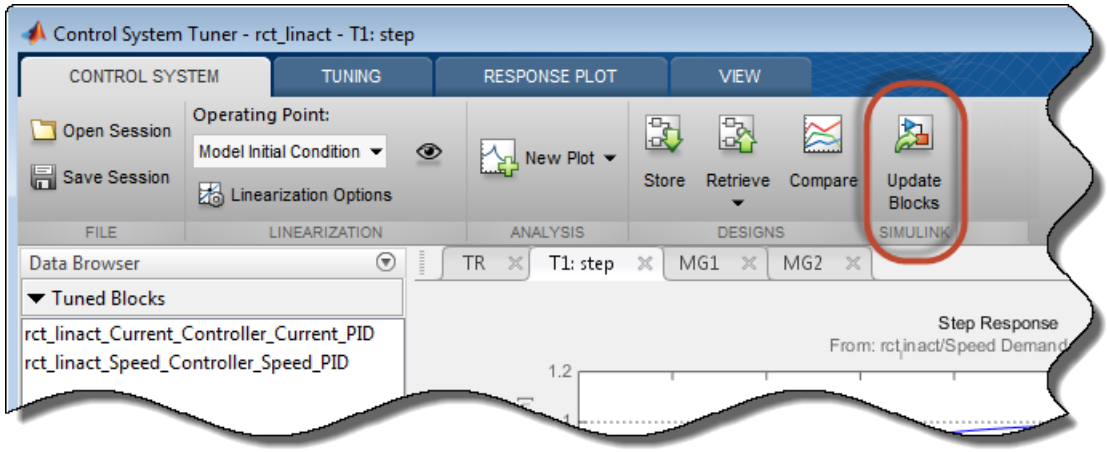
To write tuned block values from a tuned `slTuner` interface to the corresponding Simulink model, use the `writeBlockValue` command. For example, suppose `ST1` is the tuned `slTuner` interface returned by `systune`. The following command writes the tuned parameters from `ST1` to the associated Simulink model.

```
writeBlockValue(ST1)
```

Simulate the Simulink model to evaluate system performance with the tuned parameter values.

Write Parameters in Control System Tuner

To write tuned block parameters to a Simulink model, in the **Control System** tab, click  **Update Blocks**.



Control System Tuner transfers the current values of the tuned block parameters to the corresponding blocks in the Simulink model. Simulate the model to evaluate system performance using the tuned parameter values.

To update Simulink model with parameter values from a previous design stored in Control System Tuner, click **Retrieve** and select the stored design that you want to make the current design. Then click **Update Blocks**.

See Also

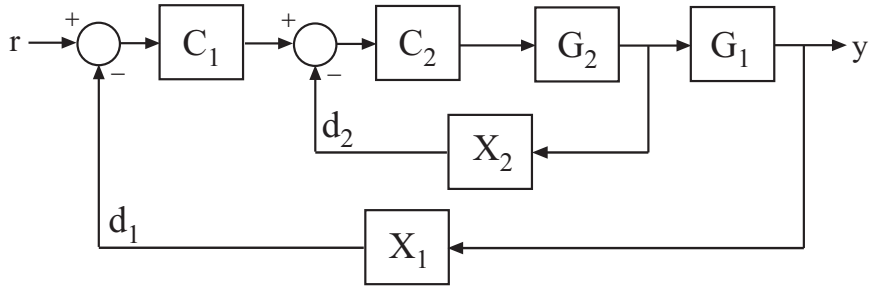
Related Examples

- "Extract Responses from Tuned MATLAB Model at the Command Line" on page 9-231
- "Create Response Plots in Control System Tuner" on page 9-197
- "Visualize Tuning Goals" on page 9-188

Extract Responses from Tuned MATLAB Model at the Command Line

This example shows how to analyze responses of a tuned control system by using `getIOTransfer` to compute responses between various inputs and outputs of a closed-loop model of the system. You can obtain other responses using similar functions such as `getLoopTransfer` and `getSensitivity`.

Consider the following control system.



Suppose you have used `systune` to tune a `genss` model of this control system. The result is a `genss` model, `T`, which contains tunable blocks representing the controller elements `C1` and `C2`. The tuned model also contains `AnalysisPoint` blocks that represent the analysis-point locations, `X1` and `X2`.

Analyze the tuned system performance by examining various system responses extracted from `T`. For example, examine the response at the output, y, to a disturbance injected at the point d_1.

```
H1 = getIOTransfer(T,'X1','y');
```

`H1` represents the closed-loop response of the control system to a disturbance injected at the implicit input associated with the `AnalysisPoint` block `X1`, which is the location of d_1:

`H1` is a `genss` model that includes the tunable blocks of `T`. `H1` allows you to validate the disturbance response of your tuned system. For example, you can use analysis commands such as `bodeplot` or `stepplot` to analyze `H1`. You can also use `getValue` to obtain the current value of `H1`, in which all the tunable blocks are evaluated to their current numeric values.

Similarly, examine the response at the output to a disturbance injected at the point d_2.

```
H2 = getIOTransfer(T,'X2','y');
```

You can also generate a two-input, one-output model representing the response of the control system to simultaneous disturbances at both d_1 and d_2. To do so, provide `getIOTransfer` with a cell array that specifies the multiple input locations.

```
H = getIOTransfer(T,{'X1','X2'},'y');
```

See Also

`AnalysisPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity`

Related Examples

- “Interpret Numeric Tuning Results” on page 9-183

Gain-Scheduled Controllers

Gain Scheduling Basics

Gain scheduling is an approach to control of nonlinear systems using a family of linear controllers, each providing satisfactory control for a different operating point of the system. Gain-scheduled control is typically implemented using a controller whose gains are automatically adjusted as a function of scheduling variables that describe the current operating point. Such variables can include time, external operating conditions, or system states such as orientation or velocity.

Gain-scheduled control systems are often designed by choosing a small set of operating points, the design points, and designing a suitable linear controller for each point. In operation, the system switches or interpolates between these controllers according to the current values of the scheduling variables.

Gain scheduling is most suitable when the scheduling variables are external parameters that vary slowly compared to the control bandwidth, such as the ambient temperature of a chemical reaction or the speed of a cruising aircraft. Gain scheduling is most challenging when the scheduling variables depend on fast-varying states of the system. Because local linear performance near operating points is no guarantee of global performance in nonlinear systems, extensive simulation-based validation is required. See [1] for an overview of gain scheduling and its challenges.

To design a gain-scheduled control system, you need:

- An operating range, defined as a set of ranges within which the values of relevant system parameters remain during operation. For instance, if your system is a cruising aircraft, then the operating range might be an incidence angle between –20° and 20° and airspeed in the range 200-250 m/s.
- Some measurable variables that indicate where in the operating range the system is at a given time. These signals are the scheduling variables. For the aircraft system, the scheduling variables might be the incidence angle and the airspeed.
- A gain schedule, which comprises the formulas or data tables that return the appropriate controller gains for given values of the scheduling variables. For the aircraft system, the gain schedule gives appropriate controller gains for any combination of incidence angle and airspeed within the operating range.

Gain Scheduling in Simulink

Control System Toolbox provides blocks that help you model gain-scheduled control systems in Simulink. These blocks let you implement common control-system elements

with variable parameters. For instance, the Varying PID Controller block accepts PID gains as inputs. In your model, you use blocks such as n-D Lookup Table or MATLAB Function blocks to implement the gain schedule. For more information and examples, see "Model Gain-Scheduled Control Systems in Simulink" on page 10-4.

Tune Gain Schedules

You can use `systune` to tune gain schedules to achieve a control system that meets performance objectives across the entire operating range. For more information, see "Tune Gain Schedules in Simulink" on page 10-15.

References

[1] Rugh, W.J., and J.S. Shamma, "Research on Gain Scheduling", *Automatica*, 36 (2000), pp. 1401-1425.

See Also

More About

- "Model Gain-Scheduled Control Systems in Simulink" on page 10-4
- "Tune Gain Schedules in Simulink" on page 10-15

Model Gain-Scheduled Control Systems in Simulink

In Simulink, you can model gain-scheduled control systems in which controller gains or coefficients depend on scheduling variables such as time, operating conditions, or model parameters. The library of linear parameter-varying blocks in Control System Toolbox lets you implement common control-system elements with variable gains. Use blocks such as lookup tables or MATLAB Function blocks to implement the gain schedule, which gives the dependence of these gains on the scheduling variables.

To model a gain-scheduled control system in Simulink:

1. Identify the scheduling variables and the signals that represent them in your model. For instance, if your system is a cruising aircraft, then the scheduling variables might be the incidence angle and the airspeed of the aircraft.
2. Use a lookup table block or a MATLAB Function block to implement a gain or coefficient that depends on the scheduling variables. If you do not have lookup table values or MATLAB expressions for gain schedules that meet your performance requirements, you can use `systune` to tune them. See “Tune Gain Schedules in Simulink” on page 10-15.
3. Replace ordinary control elements with gain-scheduled elements. For instance, instead of a fixed-coefficient PID controller, use a Varying PID Controller block, in which the gain schedules determine the PID gains.
4. Add scheduling logic and safeguards to your model as needed.

Model Scheduled Gains

A gain schedule converts the current values of the scheduling variables into controller gains. There are several ways to implement a gain schedule in Simulink.

Available blocks for implementing lookup tables include:

- Lookup tables — A lookup table is a list of breakpoints and corresponding gain values. When the scheduling variables fall between breakpoints, the lookup table interpolates between the corresponding gains. Use the following blocks to implement gain schedules as lookup tables.
  - 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table — For a scalar gain that depends on one, two, or more scheduling variables.
  - Matrix Interpolation — For a matrix-valued gain that depends on one, two, or three scheduling variables. (This block is in the **Simulink Extras** library.)

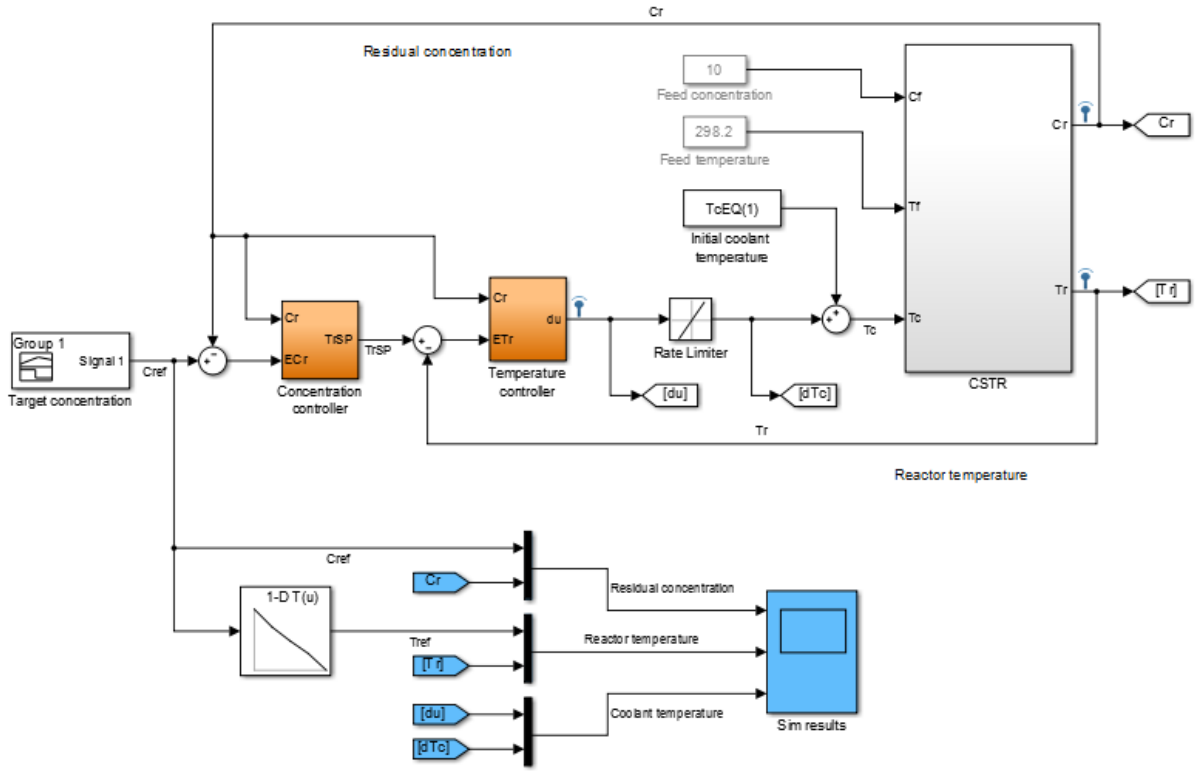
- MATLAB Function block — When you have a functional expression relating the gains to the scheduling variables, use a MATLAB Function block. If the expression is a smooth function, using a MATLAB function can result in smoother gain variations than a lookup table. Also, if you use a code-generation product such as Simulink Coderto implement the controller in hardware, a MATLAB function can result in a more memory-efficient implementation than a lookup table.

You can use `systune` to tune gain schedules implement as either lookup tables or MATLAB functions. See “Tune Gain Schedules in Simulink” on page 10-15.

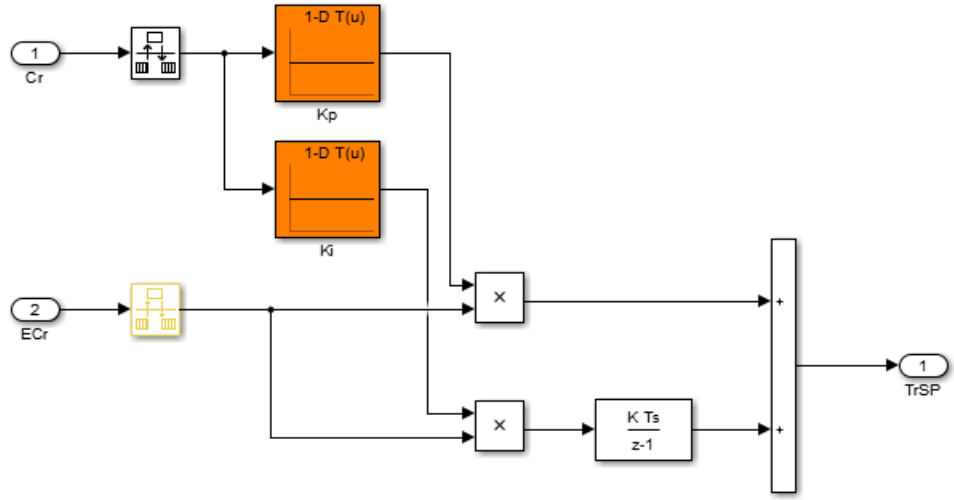
Scheduled Gain in Controller

As an example, The model `rct_CSTR` includes a PI controller and a lead compensator in which the controller gains are implemented as lookup tables using 1-D Lookup Table blocks. Open that model and examine the controllers.

```
open_system(fullfile(matlabroot,'examples','controls_id','rct_CSTR.slx'))
```



Both the `Concentration controller` and `Temperature controller` blocks take the `CSTR` plant output, `Cr`, as an input. This value is both the controlled variable of the system and the scheduling variable on which the controller action depends. Double-click the `Concentration controller` block.



Gain-scheduled PID controller Kp + Ki * Ts/(z-1)

This block is a PI controller in which the proportional gain `Kp` and integrator gain `Ki` are determined by feeding the scheduling parameter `Cr` into a 1-D Lookup Table block. Similarly, the `Temperature controller` block contains three gains implemented as lookup tables.

Gain-Scheduled Equivalents for Commonly Used Control Elements

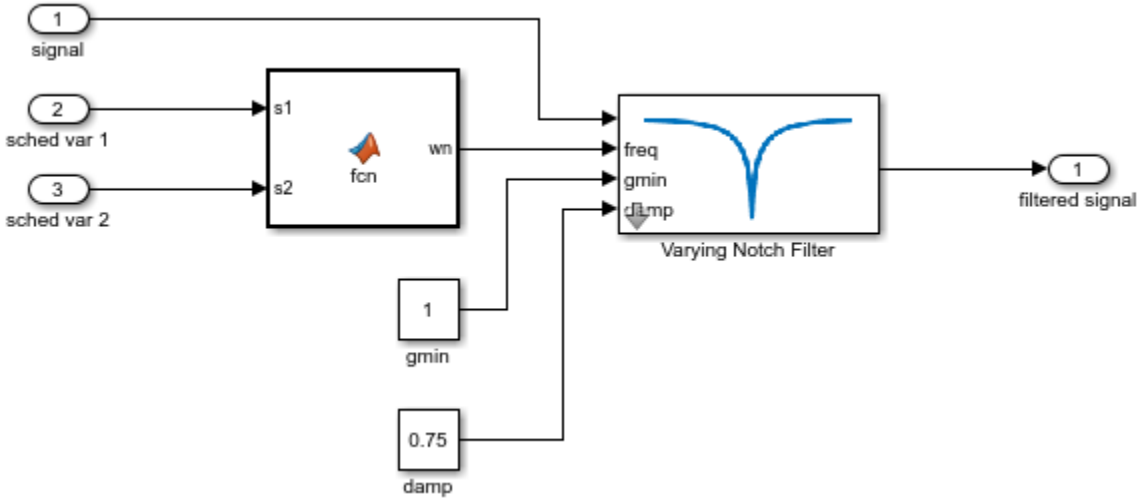
Use the **Linear Parameter Varying** block library of Control System Toolbox to implement common control elements with variable parameters or coefficients. These blocks provide common elements in which the gains or parameters are available as external inputs. The following table lists some applications of these blocks.

Block	Application
• Varying Lowpass Filter • Discrete Varying Lowpass	Use these blocks to implement a Butterworth lowpass filter in which the cutoff frequency varies with scheduling variables.

Block	Application
• Varying Notch Filter • Discrete Varying Notch	Use these blocks to implement a notch filter in which the notch frequency, width, and depth vary with scheduling variables.
• Varying PID Controller • Discrete Varying PID • Varying 2DOF PID • Discrete Varying 2DOF PID	These blocks are preconfigured versions of the PID Controller and PID Controller (2DOF) blocks. Use them to implement PID controllers in which the PID gains vary with scheduling variables.
• Varying Transfer Function • Discrete Varying Transfer Function	Use these blocks to implement a transfer function of any order in which the polynomial coefficients of the numerator and denominator vary with scheduling variables.
• Varying State Space • Discrete Varying State Space	Use these blocks to implement a state-space controller in which the *A*, *B*, *C*, and *D* matrices vary with the scheduling variables.
• Varying Observer Form • Discrete Varying Observer Form	Use these blocks to implement a gain-scheduled observer-form state-space controller, such as an LQG controller. In such a controller, the *A*, *B*, *C*, *D* matrices and the state-feedback and state-observer gain matrices vary with the scheduling variables.

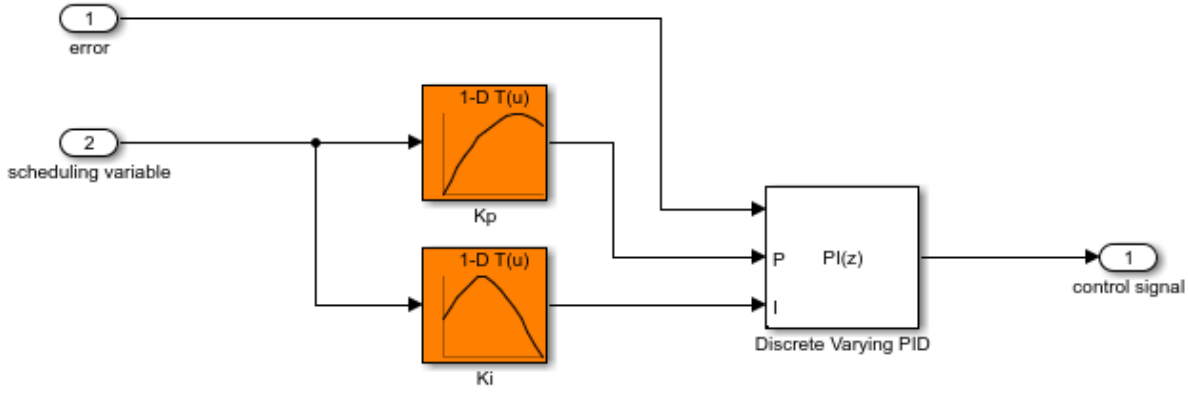
Gain-Scheduled Notch Filter

For example, the subsystem in the following illustration uses a Varying Notch Filter block to implement a filter whose notch frequency varies as a function of two scheduling variables. The relationship between the notch frequency and the scheduling variables is implemented in a MATLAB function.



Gain-Scheduled PI Controller

As another example, the following subsystem is a gain-scheduled discrete-time PI controller in which both the proportional and integral gains depend on the same scheduling variable. This controller uses 1-D Lookup Table blocks to implement the gain schedules.



Matrix-Valued Gain Schedules

You can also implement matrix-valued gain schedules Simulink. A matrix-valued gain schedule takes one or more scheduling variables and returns a matrix rather than a scalar value. For instance, suppose that you want to implement a time-varying LQG controller of the form:

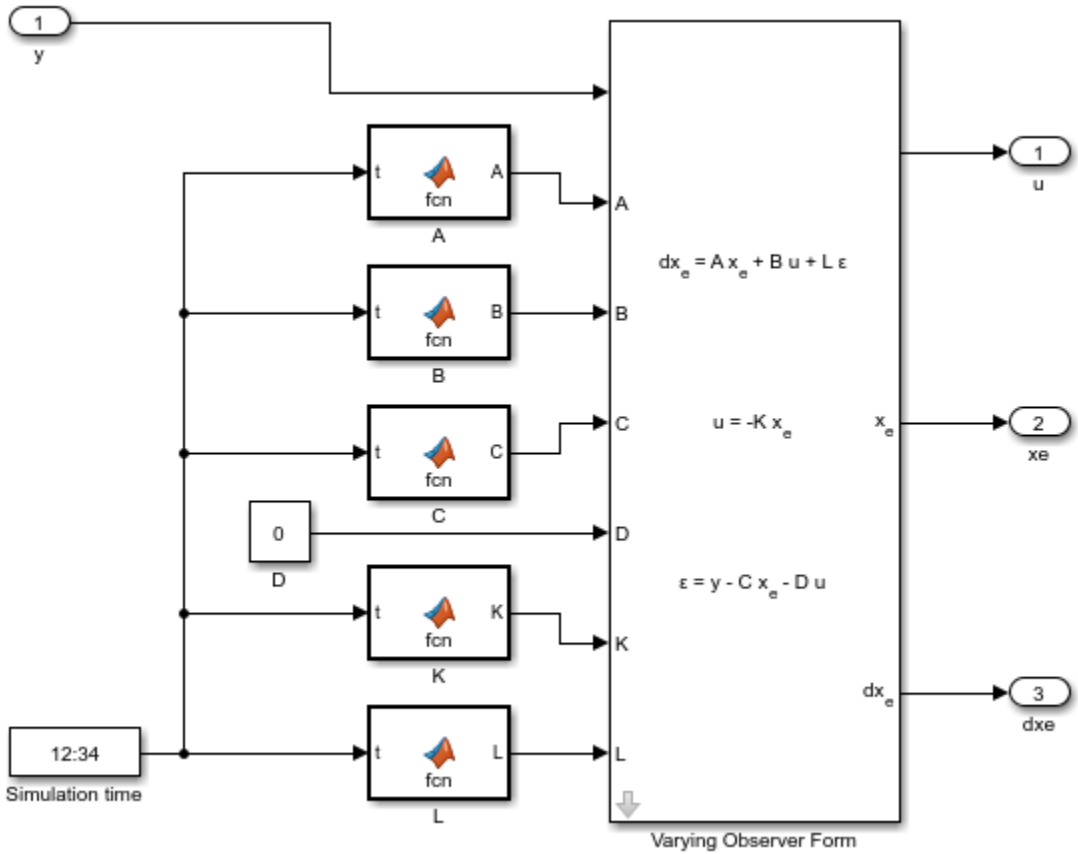
$$dx_e = Ax_e + Bu + L\left(y - Cx_e - Du\right)$$
$$u = -Kx_e,$$

where, in general, the state-space matrices A, B, C, and D, the state-feedback matrix K, and the observer-gain matrix L all vary with time. In this case, time is the scheduling variable, and the gain schedule determines the values of the matrices at a given time.

In your Simulink model, you can implement matrix-valued gain schedules using:

- MATLAB Function block — Specify a MATLAB function that takes scheduling variables and returns matrix values.
- Matrix Interpolation block — Specify a lookup table to associate a matrix value with each scheduling-variable breakpoint. Between breakpoints, the block interpolates the matrix elements. (This block is in the **Simulink Extras** library.)

For the LQG controller, use either MATLAB Function blocks or Matrix Interpolation blocks to implement the time-varying matrices as inputs to a Varying Observer Form block. For example:

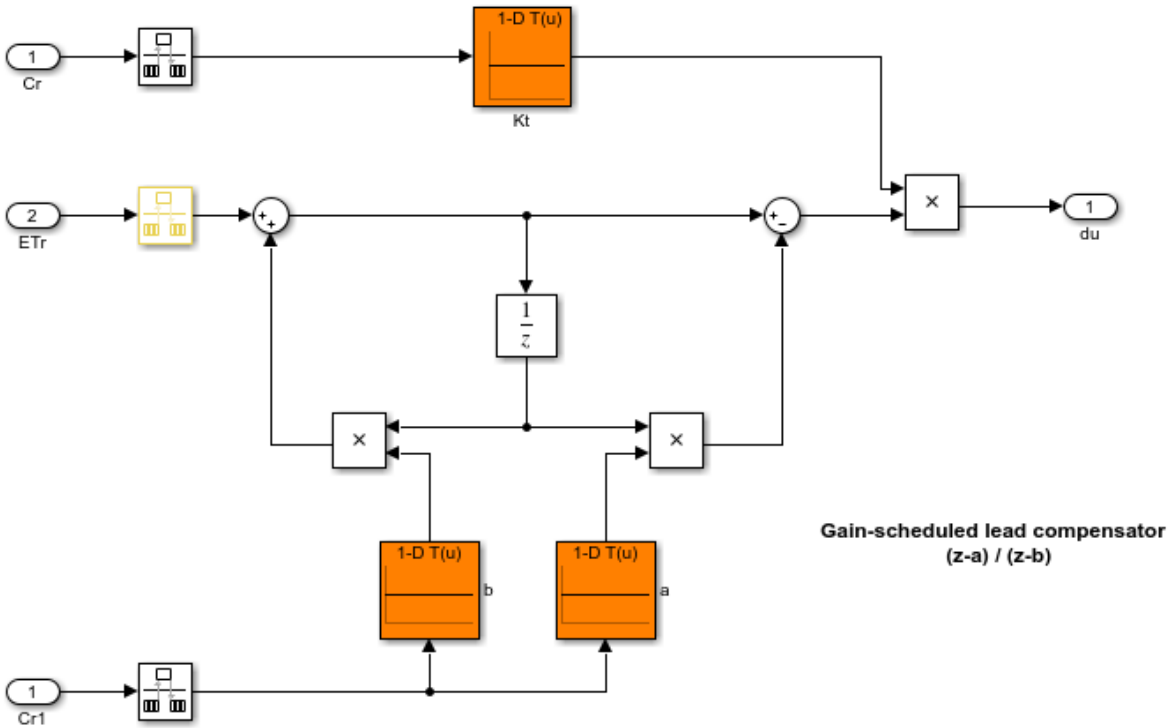


In this implementation, the time-varying matrices are each implemented as a MATLAB Function block in which the associated function takes the simulation time and returns a matrix of appropriate dimensions.

You can tune matrix-valued gain schedules implemented as either MATLAB Function blocks or as Matrix Interpolation blocks. However, to tune a Matrix Interpolation block, you must set **Simulate using** to `Interpreted execution`. See the Matrix Interpolation block reference page for information about simulation modes.

Custom Gain-Scheduled Control Structures

You can also use the scheduled gains to build your own control elements. For example, the model `rct_CSTR` includes a gain-scheduled lead compensator with three coefficients that depend on the scheduling variable, `CR`. To see how this compensator is implemented, open the model and examine the `Temperature controller` subsystem.



Here, the overall gain `Kt`, the zero location `a`, and the pole location `b` are each implemented as a 1-D lookup table that takes the scheduling variable as input. The lookup tables feed directly into product blocks.

Tunability of Gain Schedules

For a lookup table or MATLAB Function block that implements a gain schedule to be tunable with `systune`, it must ultimately feed into either:

- A block in the Linear Parameter Varying block library.
- A Product block that applies the gain to a given signal. For instance, if the Product block takes as inputs a scheduled gain $g(a)$ and a signal $u(t)$, then the output signal of the block is $y(t) = g(a)u(t)$.

There can be one or more of the following blocks between the lookup table or MATLAB Function block and the Product block or parameter-varying block:

- Gain
- Bias
- Blocks that are equivalent to a unit gain in the linear domain, including:
  - Transport Delay, Variable Transport Delay
  - Saturate, Deadzone
  - Rate Limiter, Rate Transition
  - Quantizer, Memory, Zero-Order Hold
  - MinMax
  - Data Type Conversion
  - Signal Specification
- Switch blocks, including:
  - Switch
  - Multiport Switch
  - Manual Switch

Inserting such blocks can be useful, for example, to constrain the gain value to a certain range, or to specify how often the gain schedule is updated.

See Also

Related Examples

- “Tune Gain Schedules in Simulink” on page 10-15
- “Gain-Scheduled Control of a Chemical Reactor” (Control System Toolbox)

Tune Gain Schedules in Simulink

Typically, gain-scheduled controllers are fixed single-loop or multiloop control structures in which controller gains vary with operating condition. A gain schedule converts the scheduling variables that describe the current operating condition into appropriate controller gains. In Simulink, you can implement gain schedules using lookup tables or MATLAB functions. (See "Model Gain-Scheduled Control Systems in Simulink" on page 10-4.)

You can use `systune` to tune these gain schedules so that the full nonlinear system meets your design requirements. Tuning gain schedules amounts to identifying appropriate values for lookup-table data or finding the right function to embed in a MATLAB Function block. For `systune`, you parameterize the gain schedules as functions of the scheduling variables with tunable coefficients.

Workflow for Tuning Gain Schedules

The general workflow for tuning gain-scheduled control systems is:

1. Select a set of design points that adequately covers the operating range over which you are tuning. A design point is a set of scheduling-variable values that describe a particular operating condition. The set of design points can be a regular grid of values or a scattered set. Typically, you start with a few design points. If the performance that your tuned system achieves at the design points is not maintained between design points, add more design points and retune.
2. Obtain a collection of linear models describing the linearized plant dynamics at the selected design points. Ways to obtain the array of linear models include:

   - Linearize a Simulink model at each operating condition represented in the grid of design points. For example, if each design point corresponds to a steady-state operating condition, you can trim the plant at each design point and linearize at the resulting operating point. Or, if your scheduling variable is time, you can linearize at a series of simulation snapshots.
   - Sample an LPV model of the plant at the design points.

   For more information, see "Plant Models for Gain-Scheduled Controller Tuning" on page 10-18.
3. Create an `slTuner` interface for tuning the Simulink. When you do so, you substitute the array of linear models for the plant, so that the `slTuner` interface

contains a set of closed-loop tunable models corresponding to each design point. For more information, see “Multiple Design Points in slTuner Interface” on page 10-26.

4 Model the gain schedules as parametric gain surfaces. A parametric gain surface is a basis-function expansion with tunable coefficients. For a vector σ of scheduling variables, such expansion is of the form:

$$K(\sigma) = K_0 + K_1 F_1(n(\sigma)) + \ldots + K_M F_M(n(\sigma)).$$

$n(\sigma)$ is a normalization function. For tuning with `systune`, you use `tunableSurface` to represent the parametric gain surface $K(\sigma)$. In the `slTuner` interface you create for tuning, use `setBlockParam` to associate the resulting gain surface with the block that represents the gain schedule. `systune` tunes the coefficients $K_0,\ldots,K_M$ over all the design points.

For more information, see “Parameterize Gain Schedules” on page 10-32.

5 Specify your tuning goals using `TuningGoal` objects. You can specify tuning goals that apply at all design points or at a subset of design points. You can also specify tuning goals that vary from design point to design point. For example, you might define a minimum gain margin that becomes increasingly stringent as a particular scheduling variable increases in magnitude.

For information about specifying tuning goals that vary with design point, see “Change Requirements with Operating Condition” on page 10-42.

For information about specifying tuning goals generally, see “Tuning Goals”.

6 Use `systune` to tune the control system. `systune` tunes the set of parameters, $K_0,\ldots,K_M$, against all plant models in the design grid simultaneously (multimodel tuning).

7 Validate the tuning results. You can examine the tuned gain surfaces and validate the performance of the linearized system at each design point. However, local linear performance does not guarantee global performance in nonlinear systems. Therefore, it is important to perform simulation-based validation using the tuned gain schedules.

For more information, see “Validate Gain-Scheduled Control Systems” on page 10-46.

See Also

More About

- "Model Gain-Scheduled Control Systems in Simulink" on page 10-4
- "Gain-Scheduled Control of a Chemical Reactor" (Control System Toolbox)
- "Tuning of Gain-Scheduled Three-Loop Autopilot" (Control System Toolbox)

Plant Models for Gain-Scheduled Controller Tuning

Gain scheduling is a control approach for controlling a nonlinear plant. To tune a gain-scheduled control system, you need a collection of linear models that approximate the nonlinear dynamics near selected design points. Generally, the dynamics of the plant are described by nonlinear differential equations of the form:

$$\dot{x} = f(x,u,\sigma)$$
$$y = g(x,u,\sigma).$$

Here, x is the state vector, u is the plant input, and y is the plant output. These nonlinear differential equations can be known explicitly for a particular system. More commonly, they are specified implicitly, such as by a Simulink model.

You can convert these nonlinear dynamics into a family of linear models that describe the local behavior of the plant around a family of operating points $(x(\sigma),u(\sigma))$, parameterized by the scheduling variables, σ. Deviations from the nominal operating condition are defined as:

$$\delta x = x - x(\sigma), \quad \delta u = u - u(\sigma).$$

These deviations are governed, to first order, by linear parameter-varying dynamics:

$$\dot{\delta x} = A(\sigma)\delta x + B(\sigma)\delta u, \qquad \delta y = C(\sigma)\delta x + D(\sigma)\delta u,$$

$$A(\sigma) = \frac{\partial f}{\partial x}\left(x(\sigma),u(\sigma)\right) \qquad B(\sigma) = \frac{\partial f}{\partial u}\left(x(\sigma),u(\sigma)\right)$$
$$C(\sigma) = \frac{\partial g}{\partial x}\left(x(\sigma),u(\sigma)\right) \qquad D(\sigma) = \frac{\partial g}{\partial u}\left(x(\sigma),u(\sigma)\right).$$

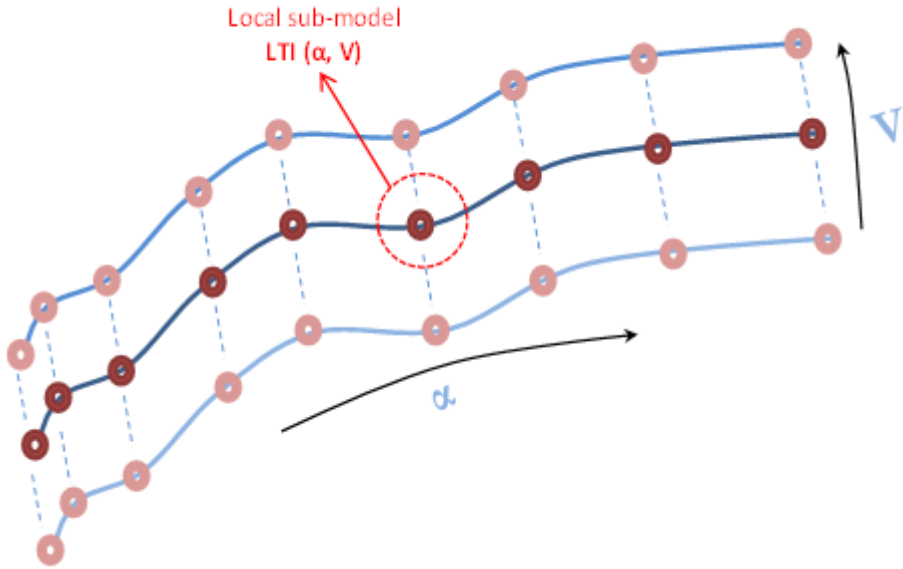
This continuum of linear approximations to the nonlinear dynamics is called a linear parameter-varying (LPV) model:

$$\frac{dx}{dt} = A(\sigma)x + B(\sigma)u$$
$$y = C(\sigma)x + D(\sigma)u.$$

The LPV model describes how the linearized plant dynamics vary with time, operating condition, or any other scheduling variable. For example, the pitch axis dynamics of an

aircraft can be approximated by an LPV model that depends on incidence angle, α, air speed, V, and altitude, h.

In practice, you replace this continuum of plant models by a finite set of linear models obtained for a suitable grid of σ values This replacement amounts to sampling the LPV dynamics over the operating range and selecting a representative set of σ values, your design points.



Gain-scheduled controllers yield best results when the plant dynamics vary smoothly between design points.

Obtaining the Family of Linear Models

If you do not have this family of linear models, there are several approaches to obtaining it, including:

- If you have a Simulink model, trim and linearize the model at the design points on page 10-20.
- Linearize the Simulink model using parameter variation on page 10-23.

- If the scheduling variable is time, linearize the model at a series of simulation snapshots on page 10-23.
- If you have nonlinear differential equations that describe the plant, linearize them at the design points.

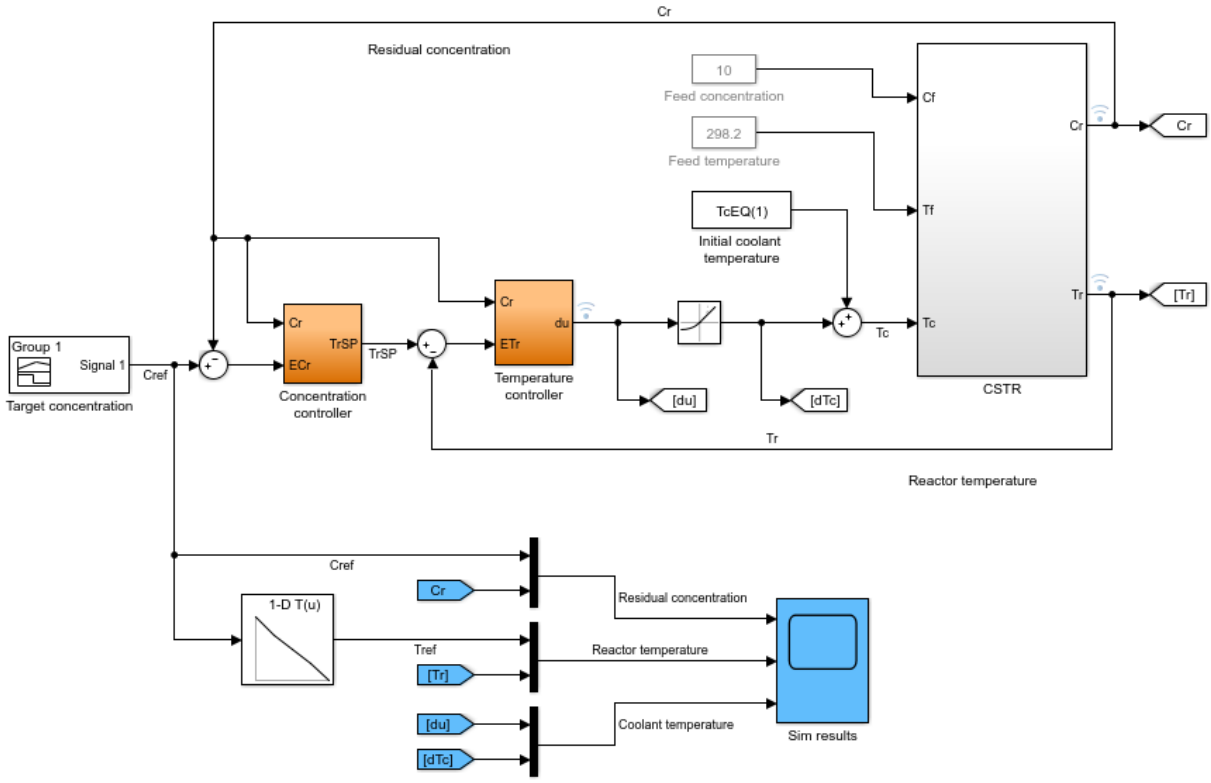
For tuning gain schedules, after you obtain the family of linear models, you must associate it with an `slTuner` interface to build a family of tunable closed-loop models. To do so, use block substitution, as described in "Multiple Design Points in slTuner Interface" on page 10-26.

Set Up for Gain Scheduling by Linearizing at Design Points

This example shows how to linearize a plant model at a set of design points for tuning of a gain-scheduled controller. The example then uses the resulting linearized models to configure an `slTuner` interface for tuning the gain schedule.

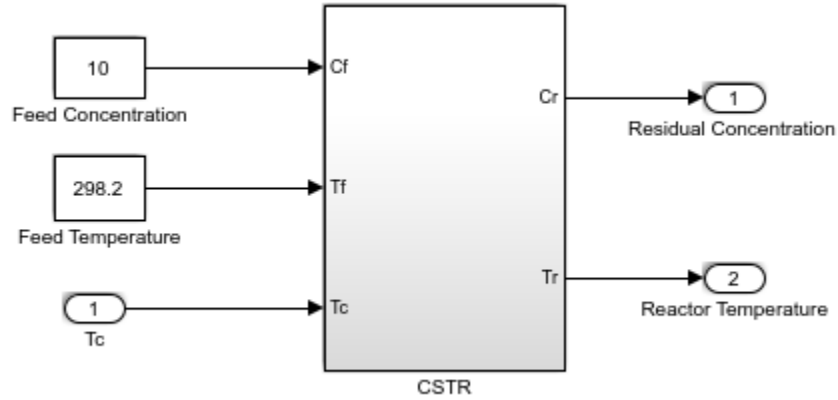
Open the `rct_CSTR` model.

```
mdl = fullfile(matlabroot,'examples','controls_id','rct_CSTR.slx');
open_system(mdl)
```



In this model, the `Concentration controller` and `Temperature controller` both depend on the output concentration `Cr`. To set up this gain-scheduled system for tuning, you linearize the plant at a set of steady-state operating points that correspond to different values of the scheduling parameter `Cr`. Sometimes, it is convenient to use a separate model of the plant for trimming and linearization under various operating conditions. For example, in this case, the most straightforward way to obtain these linearizations is to use a separate open-loop model of the plant, `rct_CSTR_OL`.

```
mdl_OL = fullfile(matlabroot,'examples','controls_id','rct_CSTR_OL.slx');
open_system('rct_CSTR_OL')
```



Trim Plant at Design Points

Suppose that you want to control this plant at a range of `Cr` values from 4 to 8. Trim the model to find steady-state operating points for a set of values in this range. These values are the design points for tuning.

```
Cr = (4:8)';        % concentrations
for k=1:length(Cr)
   opspec = operspec('rct_CSTR_OL');
   % Set desired residual concentration
   opspec.Outputs(1).y = Cr(k);
   opspec.Outputs(1).Known = true;
   % Compute equilibrium condition
   [op(k),report(k)] = findop('rct_CSTR_OL',opspec,findopOptions('DisplayReport','off')
end
```

`op` is an array of steady-state operating points. (For more information about steady-state operating points, see “About Operating Points” on page 1-2 in the Simulink Control Design documentation.)

Linearize at Design Points

Linearizing the plant model using `op` returns an array of LTI models, each linearized at the corresponding design point.

```
G = linearize('rct_CSTR_OL','rct_CSTR_OL/CSTR',op);
```

Create `slTuner` Interface with Block Substitution

To tune the control system `rct_CSTR`, create an `slTuner` interface that linearizes the system at those design points. Use block substitution to replace the plant in `rct_CSTR` with the linearized plant-model array `G`.

```
blocksub.Name = 'rct_CSTR/CSTR';
blocksub.Value = G;
tunedblocks = {'Kp','Ki'};
ST0 = slTuner('rct_CSTR',tunedblocks,blocksub);
```

For this example, only the PI coefficients in the `Concentration controller` are designated as tuned blocks. In general, however, `tunedblocks` lists all the blocks to tune.

For more information about using block substitution to configure an `slTuner` interface for gain-scheduled controller tuning, see "Multiple Design Points in slTuner Interface" (Control System Toolbox).

For another example that illustrates using trimming and linearization to generate a family of linear models for gain-scheduled controller tuning, see "Trimming and Linearization of the HL-20 Airframe" (Control System Toolbox).

Sample System at Simulation Snapshots

If you are controlling the system around a reference trajectory $(x(\sigma),u(\sigma))$, use snapshot linearization to sample the system at various points along the σ trajectory. Use this approach for time-varying systems where the scheduling variable is time.

To linearize a system at a set of simulation snapshots, use a vector of positive scalars as the `op` input argument of `linearize`, `slLinearizer`, or `slTuner`. These scalars are the simulation times at which to linearize the model. Use the same set of time values as the design points in tunable surfaces for the system.

Sample System at Varying Parameter Values

If the scheduling variable is a parameter in the Simulink model, you can use parameter variation to sample the control system over a parameter grid. For example, suppose that you want to tune a model named `suspension_gs` that contains two parameters, `Ks` and `Bs`. These parameters each can vary over some known range, and a controller gain in the model varies as a function of both parameters.

To set up such a model for tuning, create a grid of parameter values. For this example, let `Ks` vary from 1 – 5, and let `Bs` vary from 0.6 – 0.9.

```
Ks = 1:5;
Bs = [0.6:0.1:0.9];
[Ksgrid,Bsgrid] = ndgrid(Ks,Bs);
```

These values are the design points at which to sample and tune the system. For example, create an `slTuner` interface to the model, assuming one tunable block, a Lookup Table block named `K` that models the parameter-dependent gain.

```
params(1) = struct('Name','Ks','Value',Ksgrid);
params(2) = struct('Name','Bs','Value',Bsgrid);
STO = slTuner('suspension_gs','K',params);
```

`slTuner` samples the model at all `(Ksgrid,Bsgrid)` values specified in `params`.

Next, use the same design points to create a tunable gain surface for parameterizing `K`.

```
design = struct('Ks',Ksgrid,'Bs',Bsgrid);
shapefcn = @(Ks,Bs)[Ks,Bs,Ks*Bs];
K = tunableSurface('K',1,design,shapefcn);
setBlockParam(ST0,'K',K);
```

After you parameterize all the scheduled gains, you can create your tuning goals and tune the system with `systune`.

Eliminate Samples at Unneeded Design Points

Sometimes, your sampling grid includes points that do not represent or physical design points. You can eliminate such design points from the model grid entirely, so that they do not contribute to any stage of tuning or analysis. To do so, use `voidModel`, which replaces specified models in a model array with `NaN`. `voidModel` replaces specified models in a model array with `NaN`. Using `voidModel` lets your design over a grid of design points that is almost regular.

There are other tools for controlling which models contribute to design and analysis. For instance, you might want to:

- Keep a model in the grid for analysis, but exclude it from tuning.
- Keep a model in the grid for tuning, but exclude it from a particular design goal.

For more information, see “Change Requirements with Operating Condition” on page 10-42.

LPV Plants in MATLAB

In MATLAB, you can use an array of LTI plant models to represent an LPV system sampled at varying values of σ. To associate each linear model in the set with the underlying design points, use the `SamplingGrid` property of the LTI model array σ. One way to obtain such an array is to create a parametric generalized state-space (`genss`) model of the system and sample the model with parameter variation to generate the array. For an example, see “Study Parameter Variation by Sampling Tunable Model” (Control System Toolbox).

See Also

`findop` | `slTuner` | `voidModel`

Related Examples

- “Parameterize Gain Schedules” on page 10-32
- “Tune Gain Schedules in Simulink” on page 10-15
- “Multiple Design Points in slTuner Interface” on page 10-26

Multiple Design Points in slTuner Interface

For tuning a gain-scheduled control system, you must make your Simulink model linearize to an array of LTI models corresponding to the various operating conditions that are your design points. Thus, after you obtain a family of linear plant models as described in “Plant Models for Gain-Scheduled Controller Tuning” on page 10-18, you must associate it with the `slTuner` interface to your Simulink model. To do so, you use block substitution to cause `slTuner` replace the plant subsystem of the model with the array of linear models. This process builds a family of tunable closed-loop models within the `slTuner` interface.

Block Substitution for Plant

Suppose that you have an array of linear plant models obtained at each operating point in your design grid. In the most straightforward case, the following conditions are met:

- The linear models in the array correspond exactly to the plant subsystem in your model.
- Other than the elements you want to tune, nothing else in the model varies with the scheduling variables.

For a Simulink model `mdl` containing plant subsystem `G`, and a linear model array `Garr` that represents the plant at a grid of design points, the following commands create an `slTuner` interface:

```
BlockSubs = struct('Name','mdl/G','Value',Garr);
st0 = slTuner('mdl',{'Kp','Ki'},BlockSubs);
```

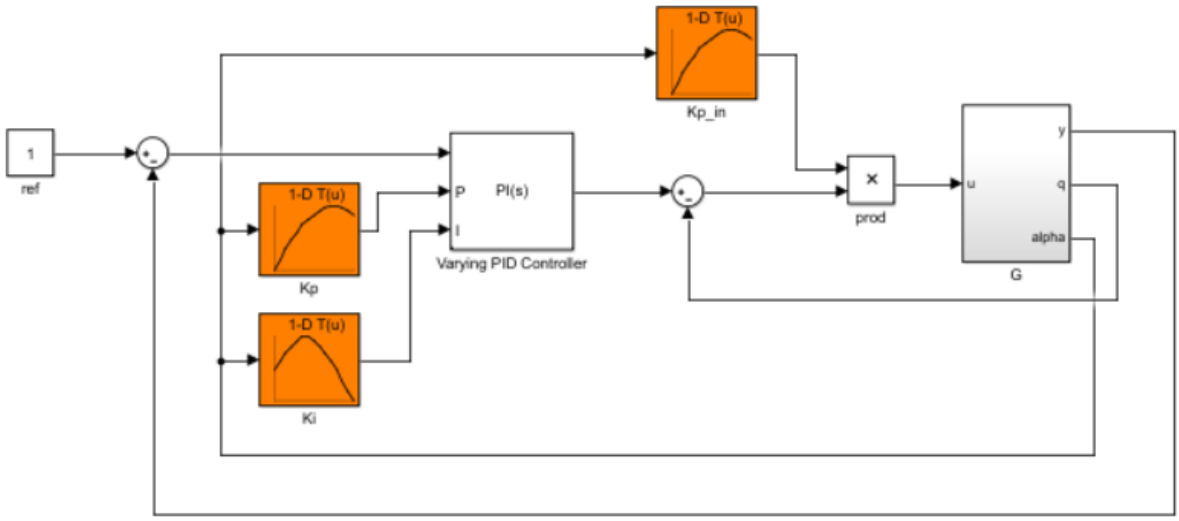
`st0` contains a family of closed-loop linear models, each linearized at a design point, and each with the corresponding linear plant inserted for `G`. If `'Kp'`and `'Ki'` are the gain schedules you want to tune (such as lookup tables), you can parameterize them with tunable gain surfaces, as described in “Parameterize Gain Schedules” on page 10-32, and tune them.

Multiple Block Substitutions

In other cases, the linearized array of plant models you have might not correspond exactly to the plant subsystem in your Simulink model. Or, you might need to replace other parts of the model that vary with operating condition. In such cases, more care is

needed in constructing the correct block substitution. The following sections highlight several such cases.

For instance, consider the model of the following illustration.



This model has an inner loop with a proportional-only gain-scheduled controller. The controller is represented by the lookup table `Kp_in` and the product block `prod`. The outer loop includes a PI controller with gain-scheduled proportional and integral coefficients represented by the lookup tables `Kp` and `Ki`. All the gain schedules depend on the same scheduling variable `alpha`.

Suppose you want to tune the inner-loop gain schedule `Kp_in` with the outer loop open. To that end, you obtain an array of linear models `G_in` from input `u` to outputs `{q,alpha}`. This model array has the wrong I/O dimensions to use as a block substitution for `G`. Therefore, you must "pad" `G_in` with an extra output dimension.

```
Garr = [0; G_in];
BlockSubs1 = struct('Name','mdl/G','Value',Garr);
```

In addition, you can remove all effect of the outer loop by replacing the Varying PID Controller block with a system that linearizes to zero at all operating conditions. Because this block has three inputs, replace it with a 3-input, one-output zero system.

```
BlockSubs2 = struct('Name','mdl/Varying PID Controller','Value',ss([0 0 0]));
```

With those block substitutions, the following commands create an `slTuner` interface that you might use to tune the inner-loop gain schedule.

```
st0 = slTuner('mdl','Kp_in');
st0.BlockSubstitutions = [BlockSubs1; BlockSubs2];
```

See the example “Angular Rate Control in the HL-20 Autopilot” (Control System Toolbox) for a another case in which several elements other than the plant itself are replaced by block substitution.

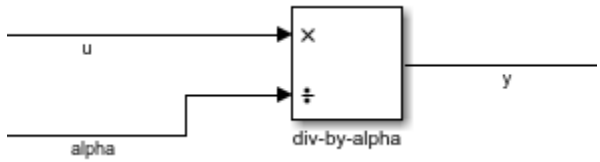
Substituting Blocks that Depend on the Scheduling Variables

Next, suppose that you have already tuned the inner-loop gain schedule, and have obtained an array `Kp_in_tuned`, of values of `Kp_in` that correspond to each design point (each value of `alpha` at which you linearized the plant). Suppose also that you have a new `Garr` that is the full plant from `u` to `{y,q,alpha}` linearized with the tuned inner loop closed. To tune the outer-loop gain schedules, you must replace the product block with the array `Kp_in_tuned`. It is important to note that you replace the injection point, the product block `prod`, rather than the lookup table `Kp_in`. Replacing the product block effectively converts it to a varying gain. Also, you must zero out the first input of the product block to remove the effect of the lookup table `Kp_in`.

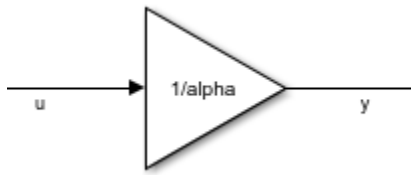
```
prodsub = [0 ss(Kp_in_tuned)];
BlockSubs1 = struct('Name','mdl/prod','Value',prodsub);
BlockSubs2 = struct('Name','mdl/G','Value',Garr);

st0 = slTuner('mdl',{'Kp','Ki'});
st0.BlockSubstitutions = [BlockSubs1; BlockSubs2];
```

For another example that shows this kind of substitution for a previously-tuned lookup table, see “Attitude Control in the HL-20 Autopilot - SISO Design” (Control System Toolbox).

The following illustration of a portion of a model highlights another scenario in which you might need to replace blocks that vary with the scheduling variable. Suppose the scheduling variable is `alpha`, and somewhere in your model, an signal `u` gets divided by `alpha`.



To ensure that `slTuner` linearizes this block correctly at all values of `alpha` in the design grid, you must replace it by an array of linear models, one for each `alpha` value. This block is equivalent to sending `u` through a gain of 1/`alpha`:



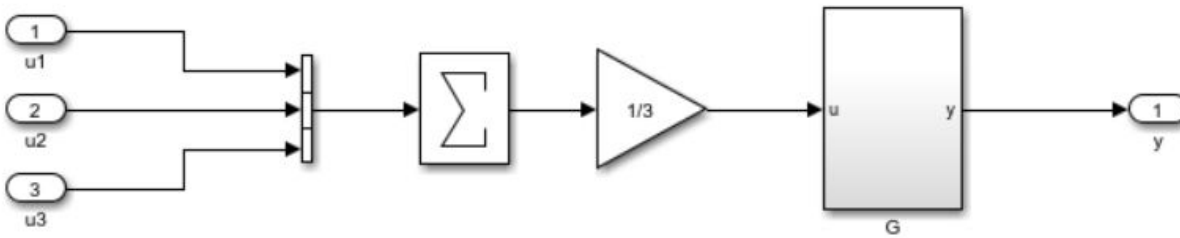
Therefore, you can use the following block substitution in your `slTuner` interface, where `alphagrid` is an array of `alpha` values at your design points.

```
divsub = ss[(1/alphagrid), 0]
BlockSubs = struct('Name','mdl/div-by-alpha','Value',divsub);
st0.BlockSubstitutions = [st0.BlockSubstitutions; BlockSubs]
```

Each entry in model array `divsub` divides its first input by the corresponding entry in `alphagrid`, and zeros out its second input. Thus, this substitution gives the desired result `y = u/alpha`.

Resolving Mismatches Between a Block and its Substitution

Sometimes, the linear model array you have is not an exact replacement for the part of the model you want to replace. For example, consider the following illustration of a three-input, one-output subsystem.



G_full

Suppose you have an array of linearized models `Garr` corresponding to `G`. You can configure a block substitution for the entire subsystem `G_full` by constructing a substitution model that reproduces the effect of averaging the three inputs, as follows:

```
Gsub = Garr*[1/3 1/3 1/3];
BlockSubs = struct('Name','mdl/G_full','Value',Gsub);
```

Sometimes, you can resolve a mismatch in I/O dimensions by padding inputs or outputs with zeros, as shown in “Multiple Block Substitutions” on page 10-26. In still other cases, you might need to perform other model arithmetic, using commands like `series`, `feedback`, or `connect` to build a suitable replacement.

Block Substitution for LPV Blocks

If the plant in your Simulink model is represented by an LPV System , you must still perform block substitution when creating the `slTuner` interface for tuning gain schedules. `slTuner` cannot read the linear model array directly from the LPV System block. However, you can use the linear model array specified in the block for the block substitution, if it corresponds to the design points for which you are tuning. For instance, suppose your plant is an LPV System block, `LPVPlant`, that specifies a model array `PlantArray`. You can configure a block substitution for `LPVPlant` as follows:

```
BlockSubs = struct('Name','mdl/LPVPlant','Value',PlantArray);
```

See Also

`slTuner`

More About

- "Tune Gain Schedules in Simulink" on page 10-15
- "Plant Models for Gain-Scheduled Controller Tuning" on page 10-18
- "Parameterize Gain Schedules" on page 10-32

Parameterize Gain Schedules

Typically, gain-scheduled control systems in Simulink use lookup tables or MATLAB Function blocks to specify gain values as a function of the scheduling variables. For tuning, you replace these blocks by parametric gain surfaces. A parametric gain surface is a basis-function expansion whose coefficients are tunable. For example, you can model a time-varying gain $k(t)$ as a cubic polynomial in t:
<mathphrase>$k(t) = k_0 + k_1 t + k_2 t^2 + k_3 t^3$.</mathphrase>

Here, $k_0,...,k_3$ are tunable coefficients. When you parameterize scheduled gains in this way, `systune` can tune the gain-surface coefficients to meet your control objectives at a representative set of operating conditions. For applications where gains vary smoothly with the scheduling variables, this approach provides explicit formulas for the gains, which the software can write directly to MATLAB Function blocks. When you use lookup tables, this approach lets you tune a few coefficients rather than many individual lookup-table entries, drastically reducing the number of parameters and ensuring smooth transitions between operating points.

Basis Function Parameterization

In a gain-scheduled controller, the scheduled gains are functions of the scheduling variables, σ. For example, a gain-scheduled PI controller has the form:

$$C(s,\sigma) = K_p(\sigma) + \frac{K_i(\sigma)}{s}.$$

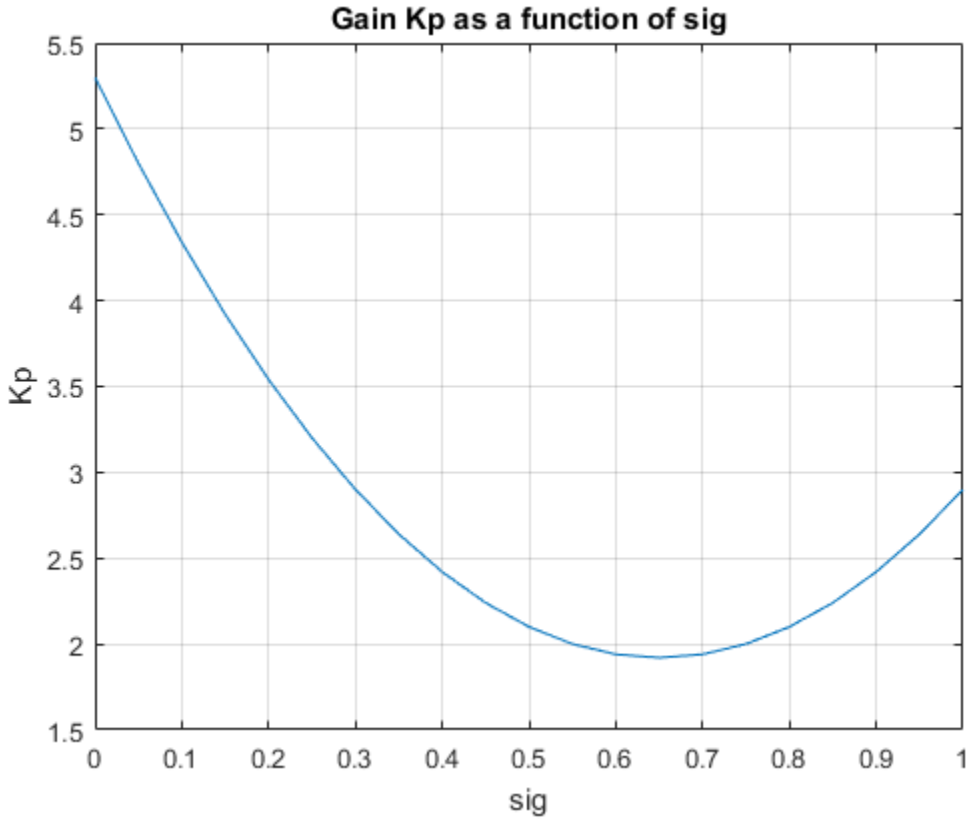
Tuning this controller requires determining the functional forms $K_p(\sigma)$ and $K_i(\sigma)$ that yield the best system performance over the operating range of σ values. However, tuning arbitrary functions is difficult. Therefore, it is necessary either to consider the function values at only a finite set of points, or restrict the generality of the functions themselves.

In the first approach, you choose a collection of design points, σ, and tune the gains K_p and K_i independently at each design point. The resulting set of gain values is stored in a lookup table driven by the scheduling variables, σ. A drawback of this approach is that tuning might yield substantially different values for neighboring design points, causing undesirable jumps when transitioning from one operating point to another.

Alternatively, you can model the gains as smooth functions of σ, but restrict the generality of such functions by using specific basis function expansions. For example, suppose σ is a scalar variable. You can model $K_p(\sigma)$ as a quadratic function of σ:

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$

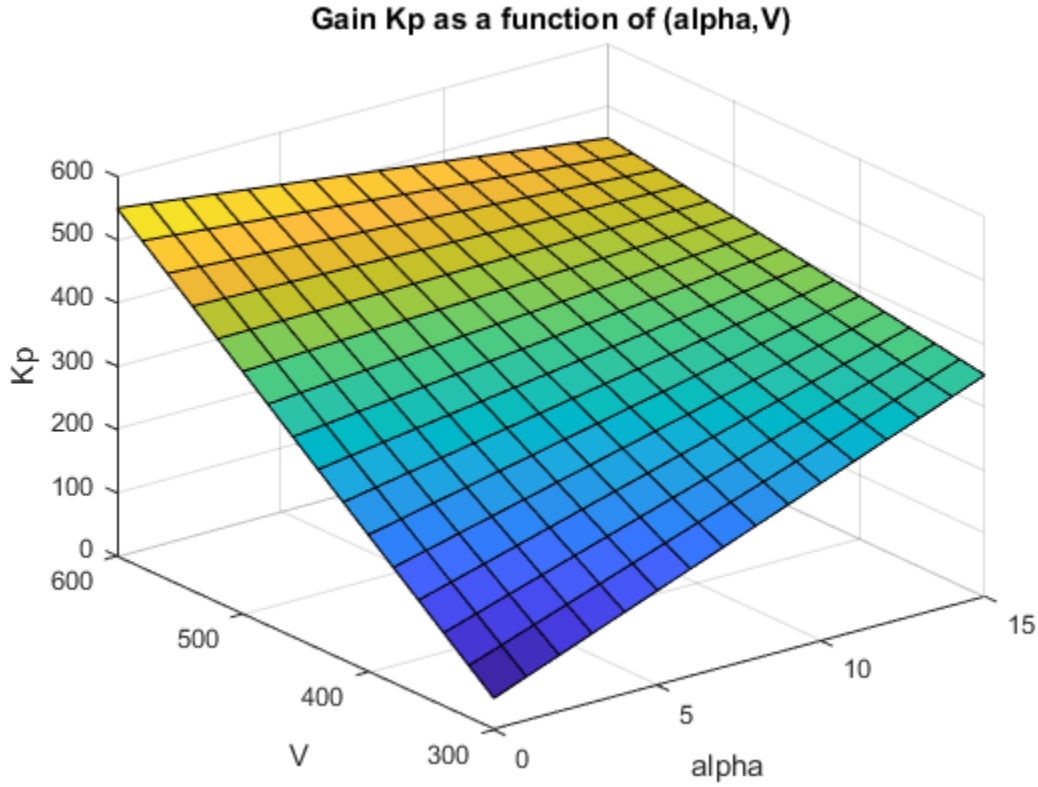
After tuning, this parametric gain might have a profile such as the following (the specific shape of the curve depends on the tuned coefficient values and range of σ):



Or, suppose that σ consists of two scheduling variables, α and V. Then, you can model $K_p(\sigma)$ as a bilinear function of α and V:

$$K_p(\alpha, V) = k_0 + k_1\alpha + k_2V + k_3\alpha V.$$

After tuning, this parametric gain might have a profile such as the following. Here too, the specific shape of the curve depends on the tuned coefficient values and ranges of σ values:



For tuning gain schedules with `systune`, you use a parametric gain surface that is a particular expansion of the gain in basis functions of σ:

$$K(\sigma) = K_0 + K_1 F_1(n(\sigma)) + \ldots + K_M F_M(n(\sigma)).$$

The basis functions $F_1,\ldots,F_M$ are user-selected and fixed. These functions operate on $n(\sigma)$, where n is a function that scales and normalizes the scheduling variables to the interval [–1,1] (or an interval you specify). The coefficients of the expansion, $K_0,\ldots,K_M$, are the tunable parameters of the gain surface. $K_0,\ldots,K_M$ can be scalar or matrix-valued, depending on the I/O size of the gain $K(\sigma)$. The choice of basis function is problem-dependent, but in general, try low-order polynomial expansions first.

Tunable Gain Surfaces

Use the `tunableSurface` command to construct a tunable model of a gain surface sampled over a grid of design points (σ values). For example, consider the gain with bilinear dependence on two scheduling variables, α and V:

$$K_p(\alpha,V) = K_0 + K_1\alpha + K_2V + K_3\alpha V.$$

Suppose that α is an angle of incidence that ranges from 0° to 15°, and V is a speed that ranges from 300 m/s to 700 m/s. Create a grid of design points that span these ranges. These design points must match the parameter values at which you sample your varying or nonlinear plant. (See “Plant Models for Gain-Scheduled Controller Tuning” on page 10-18.)

```
[alpha,V] = ndgrid(0:5:15,300:100:700);
domain = struct('alpha',alpha,'V',V);
```

Specify the basis functions for the parameterization of this surface, α, V, and αV. The `tunableSurface` command expects the basis functions to be arranged as a vector of functions of two input variables. You can use an anonymous function to express the basis functions.

```
shapefcn = @(alpha,V)[alpha,V,alpha*V];
```

Alternatively, use `polyBasis`, `fourierBasis`, or `ndBasis` to generate basis functions of as many scheduling variables as you need.

Create the tunable surface using the design points and basis functions.

```
Kp = tunableSurface('Kp',1,domain,shapefcn);
```

`Kp` is a tunable model of the gain surface. `tunableSurface` parameterizes the surface as:

$$K_p(\alpha,V) = \bar{K}_0 + \bar{K}_1\bar{\alpha} + \bar{K}_2\bar{V} + \bar{K}_3\bar{\alpha}\bar{V},$$

where

$$\bar{\alpha} = \frac{\alpha - 7.5}{7.5}, \quad \bar{V} = \frac{V - 500}{200}.$$

The surface is expressed in terms of the normalized variables, $\bar{\alpha},\bar{V} \in [-1,1]^2$ rather than in terms of α and V. This normalization, which `tunableSurface` performs by default,

improves the conditioning of the optimization performed by `systune`. If needed, you can change the default scaling and normalization. (See `tunableSurface`).

The second input argument to `tunableSurface` specifies the initial value of the constant coefficient, K_0. By default, K_0 is the gain when all the scheduling variables are at the center of their ranges. `tunableSurface` takes the I/O dimensions of the gain surface from K_0. Therefore, you can create array-valued tunable gains by providing an array for that input.

```
Karr = tunableSurface('Karr',ones(2),domain,shapefcn);
```

`Karr` is a 2-by-2 matrix in which each entry is a bilinear function of the scheduling variables with independent coefficients.

Tunable Gain With Two Independent Scheduling Variables

This example shows how to model a scalar gain K with a bilinear dependence on two scheduling variables, α and V. Suppose that α is an angle of incidence that ranges from 0 to 15 degrees, and V is a speed that ranges from 300 to 600 m/s. By default, the normalized variables are:

$$x = \frac{\alpha - 7.5}{7.5}, \quad y = \frac{V - 450}{150}.$$

The gain surface is modeled as:

$$K(\alpha, V) = K_0 + K_1 x + K_2 y + K_3 xy,$$

where $K_0, ..., K_3$ are the tunable parameters.

Create a grid of design points, (α, V), that are linearly spaced in α and V. These design points are the scheduling-variable values used for tuning the gain-surface coefficients. They must correspond to parameter values at which you have sampled the plant.

```
[alpha,V] = ndgrid(0:3:15,300:50:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range. Put them into a structure to define the design points for the tunable surface.

```
domain = struct('alpha',alpha,'V',V);
```

Create the basis functions that describe the bilinear expansion.

```
shapefcn = @(x,y) [x,y,x*y];  % or use polyBasis('canonical',1,2)
```

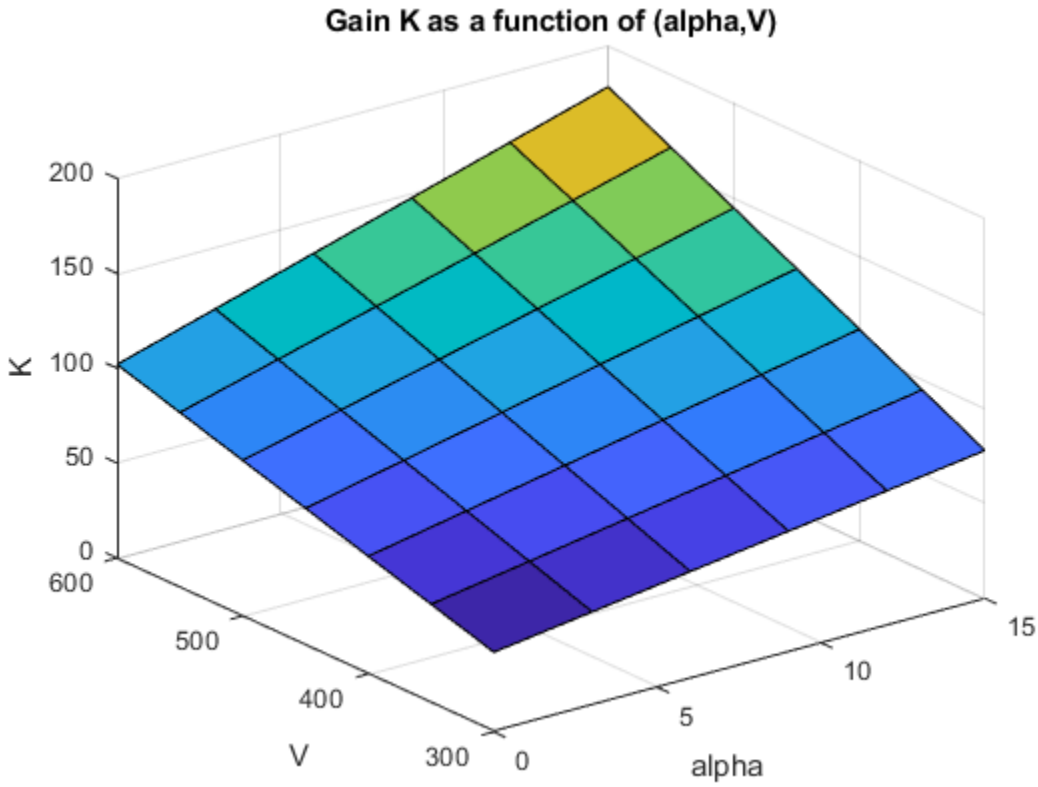
In the array returned by `shapefcn`, the basis functions are:
$F_1(x,y) = x$
$F_2(x,y) = y$
$F_3(x,y) = xy.$

Create the tunable gain surface.

```
K = tunableSurface('K',1,domain,shapefcn);
```

You can use the tunable surface as the parameterization for a lookup table block or a MATLAB Function block in a Simulink model. Or, use model interconnection commands to incorporate it as a tunable element in a control system modeled in MATLAB. After you tune the coefficients, you can examine the resulting gain surface using the `viewSurf` command. For this example, instead of tuning, manually set the coefficients to non-zero values and view the resulting gain.

```
Ktuned = setData(K,[100,28,40,10]);
viewSurf(Ktuned)
```

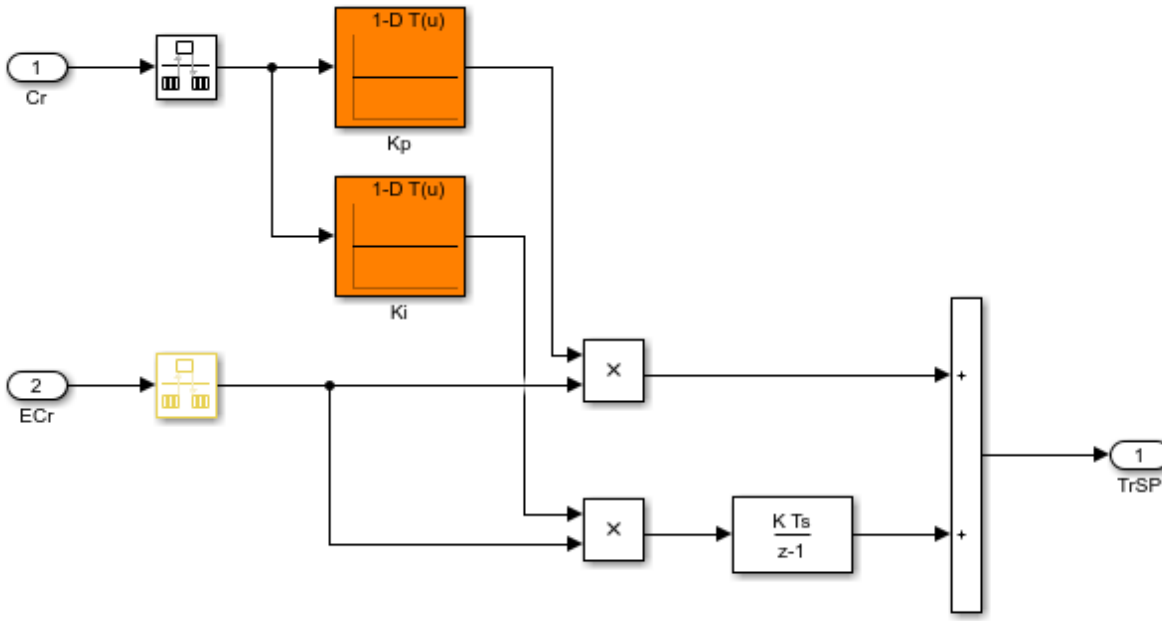


`viewSurf` displays the gain surface as a function of the scheduling variables, for the ranges of values specified by `domain` and stored in the `SamplingGrid` property of the gain surface.

Tunable Surfaces in Simulink

In your Simulink model, you model gain schedules using lookup table blocks, MATLAB Function blocks, or Matrix Interpolation blocks, as described in "Model Gain-Scheduled Control Systems in Simulink" on page 10-4. To tune these gain surfaces, use `tunableSurface` to create a gain surface for each block. In the `slTuner` interface to the model, designate each gain schedule as a block to tune, and set its parameterization to the corresponding gain surface. For instance, the `rct_CSTR` model includes a gain-

scheduled PI controller, the `Concentration controller` subsystem, in which the gains `Kp` and `Ki` vary with the scheduling variable `Cr`.



To tune the lookup tables `Kp` and `Ki`, create a tunable surface for each one. Suppose that `CrEQ` is the vector of design points, and that you expect the gains to vary quadratically with `Cr`.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp',0,TuningGrid,ShapeFcn);
Ki = tunableSurface('Ki',-2,TuningGrid,ShapeFcn);
```

Suppose that you have an array `Gd` of linearizations of the plant subsystem, `CSTR`, at each of the design points in `CrEQ`. (See “Plant Models for Gain-Scheduled Controller Tuning” on page 10-18). Create an `slTuner` interface that substitutes this array for the plant subsystem and designates the two lookup-table blocks for tuning.

```
BlockSubs = struct('Name','rct_CSTR/CSTR','Value',Gd);
ST0 = slTuner('rct_CSTR',{'Kp','Ki'},BlockSubs);
```

Finally, use the tunable surfaces to parameterize the lookup tables.

```
ST0.setBlockParam('Kp',Kp);
ST0.setBlockParam('Ki',Ki);
```

When you tune `STO`, `systune` tunes the coefficients of the tunable surfaces `Kp` and `Ki`, so that each tunable surface represents the tuned relationship between `Cr` and the gain. When you write the tuned values back to the block for validation, `setBlockParam` automatically generates tuned lookup-table data by evaluating the tunable surfaces at the breakpoints you specify in the corresponding blocks.

For more details about this example, see "Gain-Scheduled Control of a Chemical Reactor" (Control System Toolbox).

Tunable Surfaces in MATLAB

For a control system modeled in MATLAB, use tunable surfaces to construct more complex gain-scheduled control elements, such as gain-scheduled PID controllers, filters, or state-space controllers. For example, suppose that you create two gain surfaces `Kp` and `Ki` using `tunableSurface`. The following command constructs a tunable gain-scheduled PI controller.

```
C0 = pid(Kp,Ki);
```

Similarly, suppose that you create four matrix-valued gain surfaces `A`, `B`, `C`, `D`. The following command constructs a tunable gain-scheduled state-space controller.

```
C1 = ss(A,B,C,D);
```

You then incorporate the gain-scheduled controller into a generalized model of your entire control system. For example, suppose `G` is an array of models of your plant sampled at the design points that are specified in `Kp` and `Ki`. Then, the following command builds a tunable model of a gain-scheduled single-loop PID control system.

```
T0 = feedback(G*C0,1);
```

When you interconnect a tunable surface with other LTI models, the resulting model is an array of tunable generalized `genss` models. The design points in the tunable surface determine the dimensions of the array. Thus, each entry in the array represents the

system at the corresponding scheduling variable value. The `SamplingGrid` property of the array stores those design points.

```
T0 = feedback(G*Kp,1)
```

```
T0 =

  4x5 array of generalized continuous-time state-space models.
  Each model has 1 outputs, 1 inputs, 3 states, and the following blocks:
    Kp: Parametric 1x4 matrix, 1 occurrences.

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and
"T0.Blocks" to interact with the blocks.
```

The resulting generalized model has tunable blocks corresponding to the gain surfaces used to create the model. In this example, the system has one gain surface, `Kp`, which has the four tunable coefficients corresponding to K_0, K_1, K_2, and K_3. Therefore, the tunable block is a vector-valued `realp` parameter with four entries.

When you tune the control system with `systune`, the software tunes the coefficients for each of the design points specified in the tunable surface.

For an example illustrating the entire workflow in MATLAB, see the section "Controller Tuning in MATLAB" in "Gain-Scheduled Control of a Chemical Reactor" (Control System Toolbox).

See Also

`tunableSurface`

Related Examples

- "Model Gain-Scheduled Control Systems in Simulink" on page 10-4
- "Multiple Design Points in slTuner Interface" on page 10-26
- "Tune Gain Schedules in Simulink" on page 10-15

Change Requirements with Operating Condition

When tuning a gain-scheduled control system, it is sometimes useful to enforce different design requirements at different points in the design grid. For instance, you might want to:

- Specify a variable tuning goal that depends explicitly or implicitly on the design point.
- Enforce a tuning goal at a subset of design points, but ignore it at other design points.
- Exclude a design point from a particular run of `systune`, but retain it for analysis or other tuning operations.
- Eliminate a design point from all stages of design and analysis.

Define Variable Tuning Goal

There are several ways to define a tuning goal that changes across design points.

Create Varying Goals

The `varyingGoal` command lets you construct tuning goals that depend implicitly or explicitly on the design point.

For example, create a tuning goal that specifies variable gain and phase margins across a grid of design points. Suppose that you use the following 5-by-5 grid of design points to tune your controller.

```
[alpha,V] = ndgrid(linspace(0,20,5),linspace(700,1300,5));
```

Suppose further that you have 5-by-5 arrays of target gain margins and target phase margins corresponding to each of the design points, such as the following.

```
[GM,PM] = ndgrid(linspace(7,20,5),linspace(45,70,5));
```

To enforce the specified margins at each design point, first create a template for the margins goal. The template is a function that takes gain and phase margin values and returns a `TuningGoal.Margins` object with those margins.

```
FH = @(gm,pm) TuningGoal.Margins('u',gm,pm);
```

Use the template and the margin arrays to create the varying goal.

```
VG = varyingGoal(FH,GM,PM);
```

To make it easier to trace which goal applies to which design point, use the `SamplingGrid` property to attach the design-point information to `VG`.

```
VG.SamplingGrid = struct('alpha',alpha,'V',V);
```

Use `VG` with `systune` as you would use any other tuning goal. Use `viewGoal` to visualize the tuning goal and identify design points that fail to meet the target margins. For varying tuning goals, the `viewGoal` plot includes sliders that let you examine the goal and system performance for particular design points. See "Validate Gain-Scheduled Control Systems" on page 10-46.

The template function allows great flexibility in constructing the design goals. For example, you can write a function, `goalspec(a,b)`, that constructs the target overshoot as a nontrivial function of the parameters `(a,b)`, and save the function in a MATLAB file. Your template function then calls `goalspec`:

```
FH = @(a,b) TuningGoal.Overshoot('r',y',goalspec(a,b));
```

For more information about configuring varying goals, see the `varyingGoal` reference page.

Create Separate Requirement for Each Design Point

Another way to enforce a requirement that varies with design point is to create a separate instance of the requirement for each design point. This approach can be useful when you have a goal that only applies to a few of models in the design array. For example, suppose that you want to enforce a 1/*s* loop shape on the first five design points only, with a crossover frequency that depends on the scheduling variables. Suppose also that you have created a vector, `wc`, that contains the target bandwidth for each design point. Then you can construct one `TuningGoal.LoopShape` requirement for each design point. Associate each `TuningGoal.LoopShape` requirement with the corresponding design point using the `Models` property of the requirement.

```
for ct = 1:length(wc)
   R(ct) = TuningGoal.LoopShape('u',wc(ct));
   R(ct).Model = ct;
end
```

If `wc` covers all the design points in your grid, this approach is equivalent to using a `varyingGoal` object. It is a useful alternative to `varyingGoal` when you only want to constrain a few design points.

Build Variation into the Model

Instead of creating varying requirements, you can incorporate the varying portion of the requirement into the closed-loop model of the control system. This approach is a form of goal normalization that makes it possible to cover all design points with a single uniform goal.

For example, suppose that you want to limit the gain from `d` to `y` to a quantity that depends on the scheduling variables. Suppose that `T0` is an array of models of the closed-loop system at each design point. Suppose further that you have created a table, `gmax`, of the maximum gain values for each design point, σ. Then you can add another output `ys` = `y/gmax` to the closed-loop model, as follows.

```
% Create array of scalar gains 1/gmax
yScaling = reshape(1./gmax,[1 1 size(gmax)]);
yScaling = ss(yScaling,'InputName','y','OutputName','ys');

% Connect these gains in series to y output of T0
T0 = connect(T0,yScaling,T0.InputName,[T0.OutputName ; {'ys'}]);
```

The maximum gain changes at each design point according to the table `gmax`. You can then use a single requirement that limits to 1 the gain from `d` to the scaled output `ys`.

```
R = TuningGoal.Gain('d','ys',1);
```

Such effective normalization of requirements moves the requirement variability from the requirement object, `R`, to the closed-loop model, `T0`.

In Simulink, you can use a similar approach by feeding the relevant model inputs and outputs through a gain block. Then, when you linearize the model, change the gain value of the block with the operating condition. For example, set the gain to a MATLAB variable, and use the `Parameters` property in `slLinearizer` to change the variable value with each linearization condition.

Enforce Tuning Goal at Subset of Design Points

You can restrict application of a tuning goal to a subset of models in the design grid using the `Models` property of the tuning goal. Specify models by their linear index in the model array. For instance, suppose that you have a tuning goal, `Req`. Configure `Req` to apply to the first and last models in a 3-by-3 design grid.

```
Req.Models = [1,9];
```

When you call `systune` with `Req` as a hard or soft goal, `systune` enforces `Req` for these models and ignores it for the rest of the grid.

Exclude Design Points from systune Run

You can exclude one or more design points from tuning without removing the corresponding model from the array or reconfiguring your tuning goals. Doing so can be useful, for example, to identify problematic design points when tuning over the entire design grid fails to meet your design requirements. It can also be useful when there are design points that you want to exclude from a particular tuning run, but preserve for performance analysis or further tuning.

The `SkipModels` option of `systuneOptions` lets you specify models in the design grid to exclude from tuning. Specify models by their linear index in the model array. For instance, configure `systuneOptions` to skip the first and last models in a 3-by-3 design grid.

```
opt = systuneOptions;
opt.SkipModels = [1,9];
```

When you call `systune` with `opt`, the tuning algorithm ignores these models.

As an alternative, you can eliminate design points from the model grid entirely, so that they do not contribute to any stage of tuning or analysis. To do so, use `voidModel`, which replaces specified models in a model array with `NaN`. This option is useful when your sampling grid includes points that represent irrelevant or unphysical design points. Using `voidModel` lets you design over a grid of design points that is almost regular.

See Also

`systuneOptions` | `varyingGoal` | `viewGoal`

More About

- “Validate Gain-Scheduled Control Systems” on page 10-46
- “Tune Gain Schedules in Simulink” on page 10-15

Validate Gain-Scheduled Control Systems

Tuned gain schedules require careful validation. The tuning process guarantees suitable performance only near each design point. In addition, the tuning ignores dynamic couplings between the plant state variables and the scheduling variables (see Section 4.3, “Hidden Coupling”, in [1]). Best practices for validation include:

- Examine tuned gain surfaces to make sure that they are smooth and well-behaved.
- Visualize tuning goals against system responses at all design points.
- Check linear performance of the tuned control system between design points.
- Validate gain schedules in simulation of the full nonlinear system.

Check linear performance on a denser grid of σ values than you used for design. If adequate linear performance is not maintained between design points, you can add more design points and retune.

Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

Examine Tuned Gain Surfaces

After tuning, examine the tuned gains as a function of the scheduling variables to make sure that they are smooth and well-behaved over the operating range. Visualize tuned gain surfaces using the `viewSurf` command.

Visualize Tuning Goals

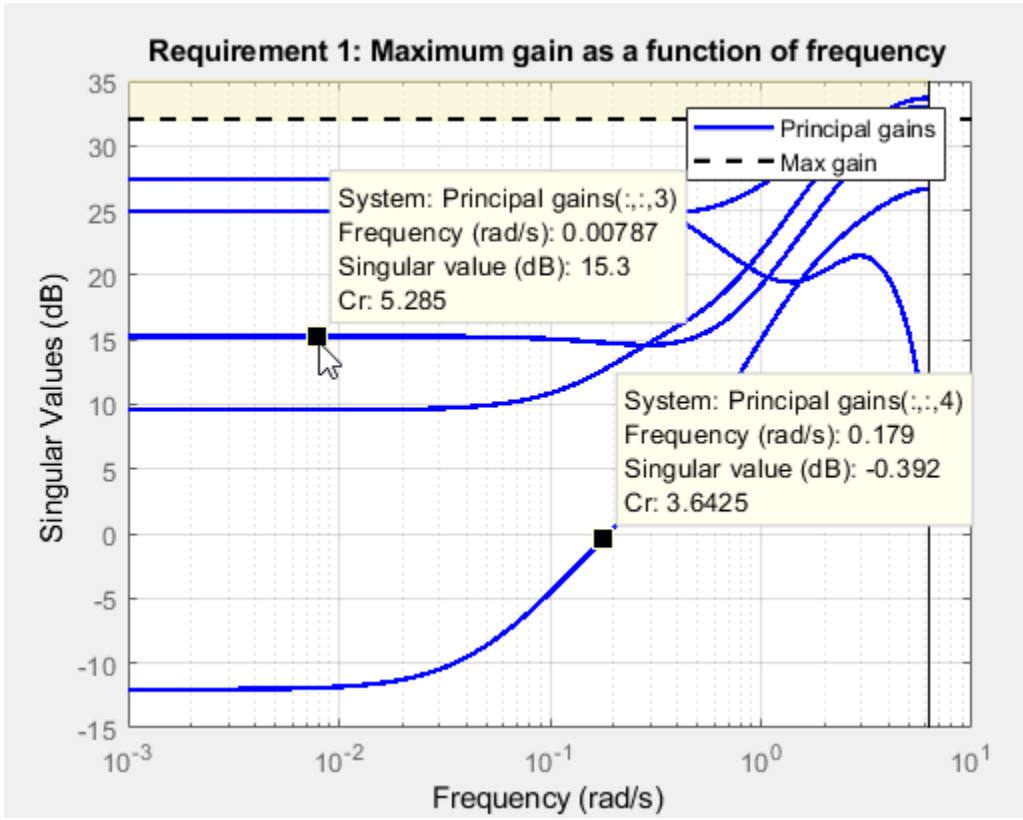
Use tuning-goal plots to visualize your design requirements against the linear response of the tuned control system. Tuning-goal plots show graphically where and by how much tuning goals are satisfied or violated. This visualization lets you examine how close your control system is to ideal performance. It can also help you identify problems with tuning and provide clues on how to improve your design.

For general information about using tuning-goal plots, see “Visualize Tuning Goals” on page 9-188. For gain-scheduled control systems, the tuning-goal plots you generate with `viewGoal` provide additional information that helps you evaluate how each tuning goal contributes to the result.

Fixed Tuning Goals

For fixed tuning goals that apply to multiple design points, `viewGoal` plots the relevant system response at all those design points. For instance, suppose that you tune an `slTuner` interface, `ST`, for the `rct_CSTR` model described in "Gain-Scheduled Control of a Chemical Reactor" (Control System Toolbox). You can use `viewGoal` to see how well each of the five design points of that example satisfies the gain goal `R3`. The resulting plot shows the relevant gain profile at all five design points. Click any of the gain lines for a display that shows the corresponding value of the scheduling variable `Cr`.

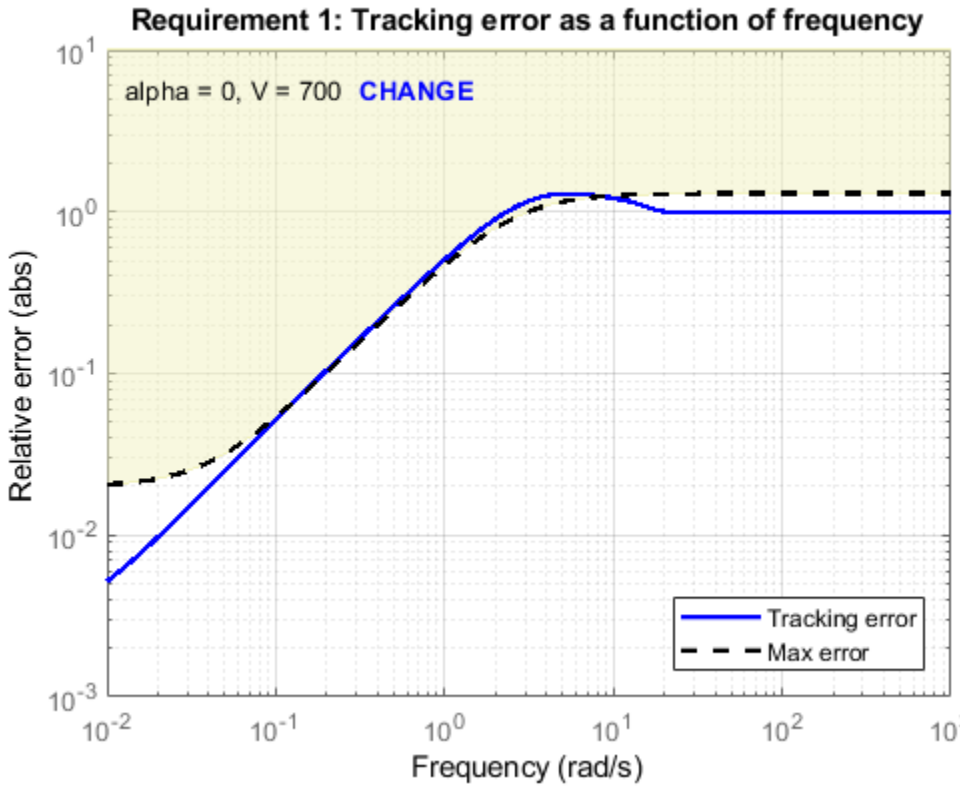
```
viewGoal(R3,ST)
```



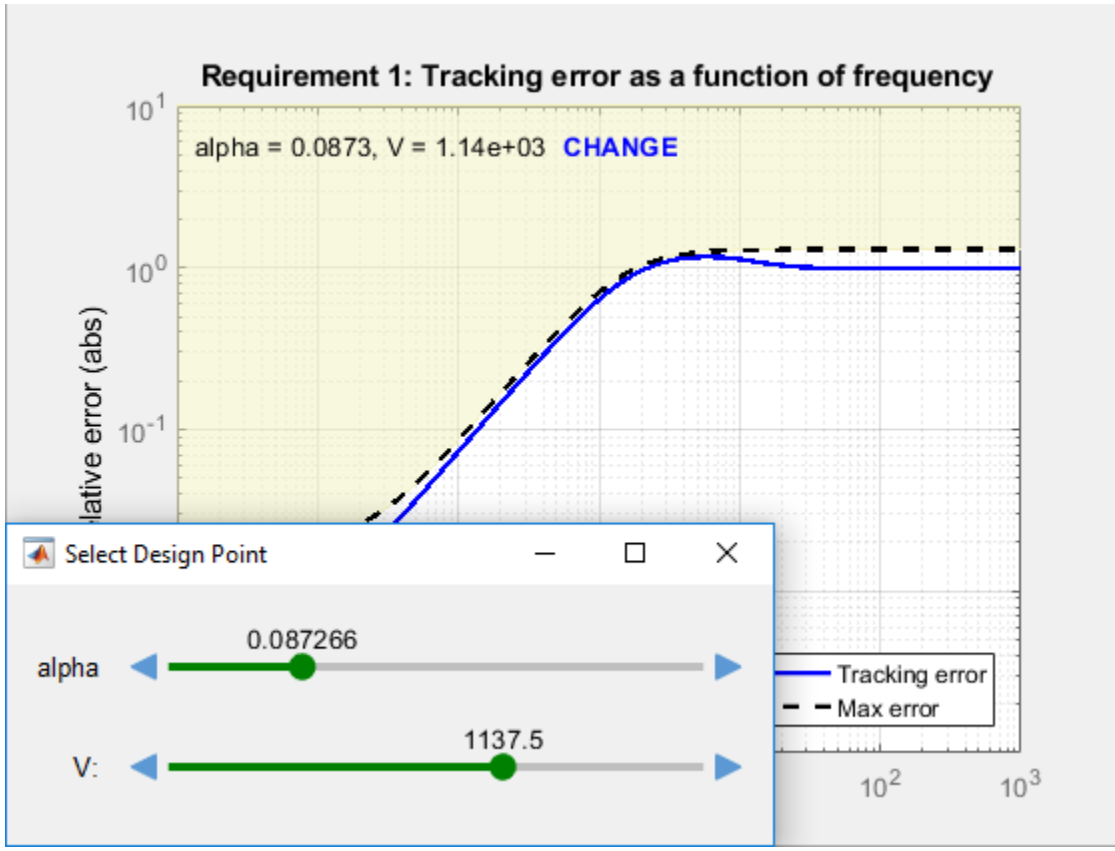
Varying Tuning Goals

Varying goals that you create using `varyingGoal` apply a different target response at each design point. When you use viewGoal to examine a varying goal, the plot initially displays the target and tuned responses at the first design point in the design grid. For instance, suppose that you tune a control system `ST` over a design grid of two scheduling variables, using a varying goal `Rv` that varies across the entire grid. After tuning, examine `Rv`.

```
viewGoal(Rv,ST)
```



Click **CHANGE** to open sliders that let you select a design point at which to view the target and tuned responses.



Check Linear Performance

In addition to examining linear responses associated with tuning goals, check other linear responses of the system to make sure that the behavior is suitable. You can do so by extracting and plotting system responses as described generally in “Validate Tuned Control System” on page 9-226.

For gain-scheduled systems, it is good practice to check linear performance on a denser grid of operating points than you used for design. If the system does not maintain adequate linear performance between design points, then you can add more design points and retune.

Validate Gain Schedules in Nonlinear System

Because `systune` tunes gain schedules against a linearization obtained at each design point, it is important to test the tuning results in simulation of the full nonlinear system. Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

After tuning an `slTuner` interface, use `writeBlockValue` to write tuned controller parameters to the Simulink model for such simulation. This command can write tuned gain schedules to lookup table blocks, Matrix Interpolation blocks, and MATLAB Function blocks for which you have specified a `tunableSurface` parameterization.

Lookup Tables

For lookup table blocks and Matrix Interpolation blocks, `writeBlockValue` automatically evaluates the tuned gain surface at the breakpoints specified in the block. These breakpoints do not need to be the same as the design points used for tuning. Because the `tunableSurface` describes the gain schedule in parametric form, `writeBlockValue` can evaluate the gain at any scheduling-variable value.

If you have retuned a subset of design points, you can use `writeLookupTableData` to update a portion of the lookup-table data while leaving the rest intact.

MATLAB Function Blocks

For gain schedules implemented as MATLAB Function blocks, `writeBlockValue` automatically generates MATLABcode and pushes it to the block. The generated MATLAB function takes the scheduling variables and returns the gain value given by the tuned parametric expression of the `tunableSurface`. To see this MATLAB code for a particular gain surface, use the `codegen` command.

References

[1] Rugh, W.J., and J.S. Shamma, “Research on Gain Scheduling”, *Automatica*, 36 (2000), pp. 1401-1425.

See Also

`codegen` | `viewGoal` | `viewSurf` | `writeBlockValue` | `writeLookupTableData`

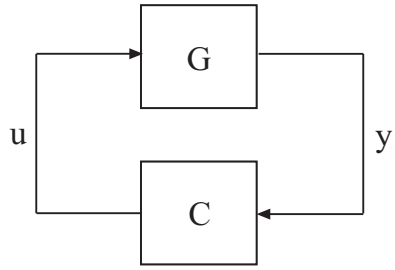
Related Examples

- “Tuning of Gain-Scheduled Three-Loop Autopilot” (Control System Toolbox)
- “Gain-Scheduled Control of a Chemical Reactor” (Control System Toolbox)
- “Validate Tuned Control System” on page 9-226

Loop-Shaping Design

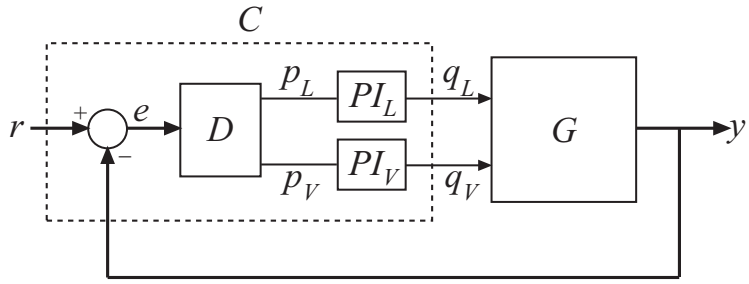
Structure of Control System for Tuning With looptune

`looptune` tunes the feedback loop illustrated below to meet default requirements or requirements that you specify.



`C` represents the controller and `G` represents the plant. The sensor outputs`y` (measurement signals) and actuator outputs `u` (control signals) define the boundary between plant and controller. The controller is the portion of your control system whose inputs are measurements, and whose outputs are control signals. Conversely, the plant is the remainder—the portion of your control system that receives control signals as inputs, and produces measurements as outputs.

For example, in the control system of the following illustration, the controller `C` receives the measurement `y`, and the reference signal `r`. The controller produces the controls `qL` and `qV` as outputs.



The controller `C` has a fixed internal structure. `C` includes a gain matrix `D` , the PI controllers `PI_L` and `PI_V`, and a summing junction. The `looptune` command tunes free parameters of `C` such as the gains in `D` and the proportional and integral gains of `PI_L` and `PI_V`. You can also use `looptune` to co-tune free parameters in both `C` and `G`.

Set Up Your Control System for Tuning with looptune

In this section...

Set Up Your Control System for looptunein MATLAB

To set up your control system in MATLAB for tuning with `looptune`:

1. Parameterize the tunable elements of your controller. You can use predefined structures such as `tunablePID`, `tunableGain`, and `tunableTF`. Or, you can create your own structure from elementary tunable parameters (`realp`).
2. Use model interconnection commands such as `series` and `connect` to build a tunable `genss` model representing the controller `C0`.
3. Create a Numeric LTI model (Control System Toolbox) representing the plant `G`. For co-tuning the plant and controller, represent the plant as a tunable `genss` model.

Set Up Your Control System for looptune in Simulink

To set up your control system in Simulink for tuning with `systune` (requires Simulink Control Design software):

1. Use `slTuner` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model.
2. Use `addPoint` to specify the control and measurement signals that define the boundaries between plant and controller. Use `addOpening` to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements.

The `slTuner` interface automatically linearizes your Simulink model. The `slTuner` interface also automatically parametrizes the blocks that you specify as tunable blocks. For more information about this linearization, see the `slTuner` reference page and “How Tuned Simulink Blocks Are Parameterized” on page 9-37.

See Also

Related Examples

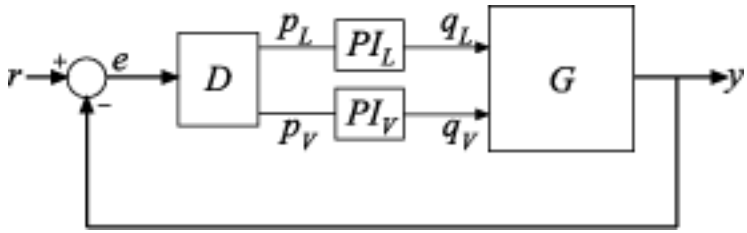
- “Tune MIMO Control System for Specified Bandwidth” on page 11-5
- “Tuning Feedback Loops with LOOPTUNE” (Control System Toolbox)

More About

- “Structure of Control System for Tuning With looptune” on page 11-2

Tune MIMO Control System for Specified Bandwidth

This example shows how to tune the following control system to achieve a loop crossover frequency between 0.1 and 1 rad/s, using `looptune`.



The plant, `G`, is a two-input, two-output model (*y* is a two-element vector signal). For this example, the transfer function of `G` is given by:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

This sample plant is based on the distillation column described in more detail in the example "Decoupling Controller for a Distillation Column" (Control System Toolbox).

To tune this control system, you first create a numeric model of the plant. Then you create tunable models of the controller elements and interconnect them to build a controller model. Then you use `looptune` to tune the free parameters of the controller model. Finally, examine the performance of the tuned system to confirm that the tuned controller yields desirable performance.

Create a model of the plant.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = 'y';
```

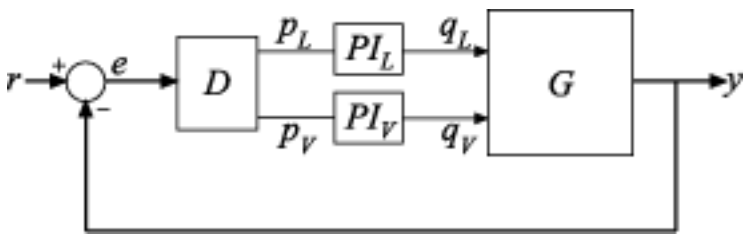
When you tune the control system, `looptune` uses the channel names `G.InputName` and `G.OutputName` to interconnect the plant and controller. Therefore, assign these channel names to match the illustration. When you set `G.OutputName = 'y'`, the `G.OutputName` is automatically expanded to `{'y(1)';'y(2)'}`. This expansion occurs because `G` is a two-output system.

Represent the components of the controller.

```
D = tunableGain('Decoupler',eye(2));
D.InputName = 'e';
D.OutputName = {'pL','pV'};

PI_L = tunablePID('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';

PI_V = tunablePID('PI_V','pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';

sum1 = sumblk('e = r - y',2);
```

The control system includes several tunable control elements. `PI_L` and `PI_V` are tunable PI controllers. These elements represented by `tunablePID` models. The fixed control structure also includes a decoupling gain matrix `D`, represented by a tunable `tunableGain` model. When the control system is tuned, `D` ensures that each output of `G` tracks the corresponding reference signal `r` with minimal crosstalk.

Assigning `InputName` and `OutputName` values to these control elements allows you to interconnect them to create a tunable model of the entire controller `C` as shown.



When you tune the control system, `looptune` uses these channel names to interconnect `C` and `G`. The controller `C` also includes the summing junction `sum1`. This a two-channel summing junction, because `r` and `y` are vector-valued signals of dimension 2.

Connect the controller components.

```
C0 = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});
```

`C0` is a tunable `genss` model that represents the entire controller structure. `C0` stores the tunable controller parameters and contains the initial values of those parameters.

Tune the control system.

The inputs to `looptune` are `G` and `C0`, the plant and initial controller models that you created. The input `wc = [0.1,1]` sets the target range for the loop bandwidth. This input specifies that the crossover frequency of each loop in the tuned system fall between 0.1 and 1 rad/min.

```
wc = [0.1,1];
[G,C,gam,Info] = looptune(G,C0,wc);
```

```
Final: Peak gain = 0.9, Iterations = 24
Achieved target gain value TargetGain=1.
```

The displayed `Peak Gain = 0.949` indicates that `looptune` has found parameter values that achieve the target loop bandwidth. `looptune` displays the final peak gain value of the optimization run, which is also the output `gam`. If `gam` is less than 1, all tuning requirements are satisfied. A value greater than 1 indicates failure to meet some requirement. If `gam` exceeds 1, you can increase the target bandwidth range or relax another tuning requirement.

`looptune` also returns the tuned controller model `C`. This model is the tuned version of `C0`. It contains the PI coefficients and the decoupling matrix gain values that yield the optimized peak gain value.

Display the tuned controller parameters.

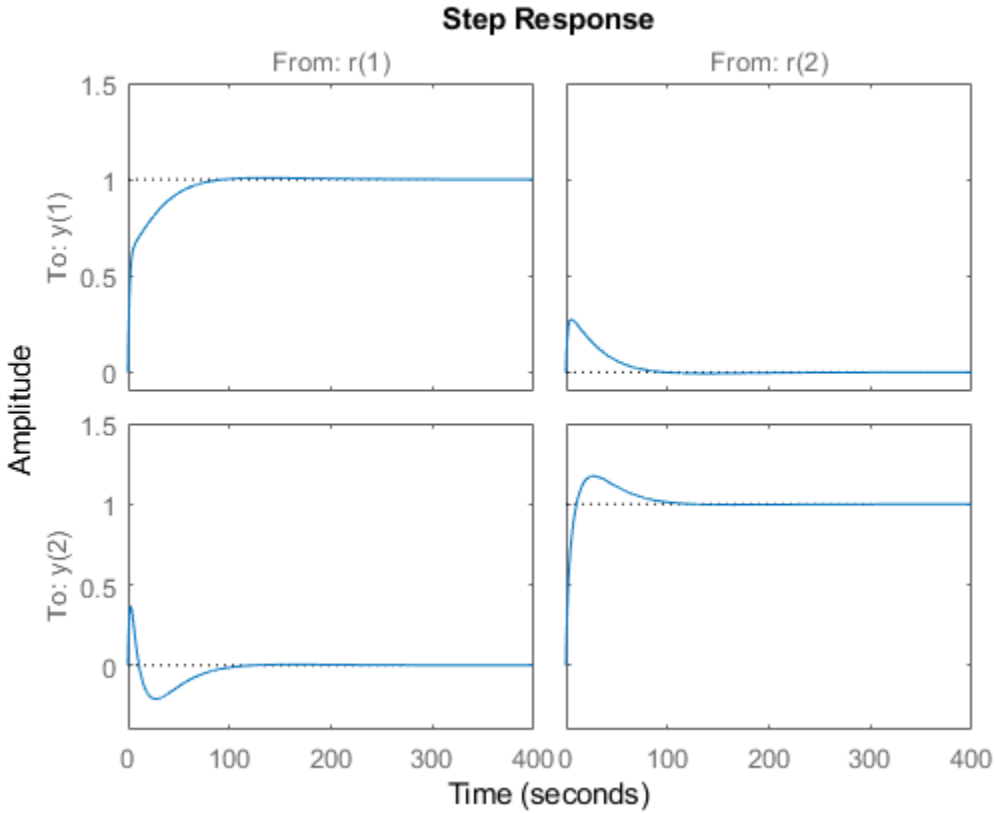
```
showTunable(C)
```

```
Decoupler =

  D =
          u1      u2
   y1   1.751  -1.327
   y2   -1.06  0.9725

Name: Decoupler
Static gain.
-----------------------------------
PI_L =

             1
  Kp + Ki * ---
             s
```

```
  with Kp = 1.82, Ki = 0.0362

Name: PI_L
Continuous-time PI controller in parallel form.
----------------------------------
PI_V =

             1
  Kp + Ki * ---
             s

  with Kp = -2.71, Ki = -0.0708

Name: PI_V
Continuous-time PI controller in parallel form.
```

Check the time-domain response for the control system with the tuned coefficients. To produce a plot, construct a closed-loop model of the tuned control system. Plot the step response from reference to output.

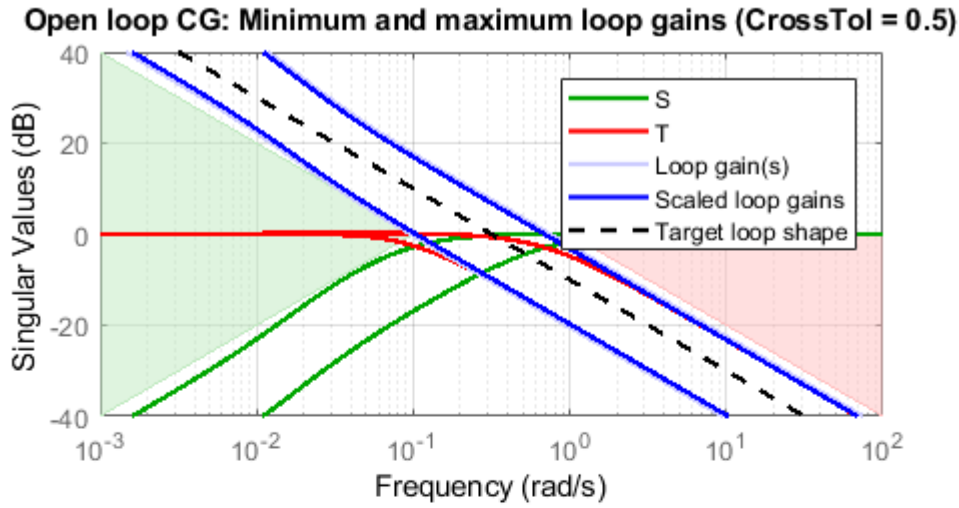
```
T = connect(G,C,'r','y');
step(T)
```

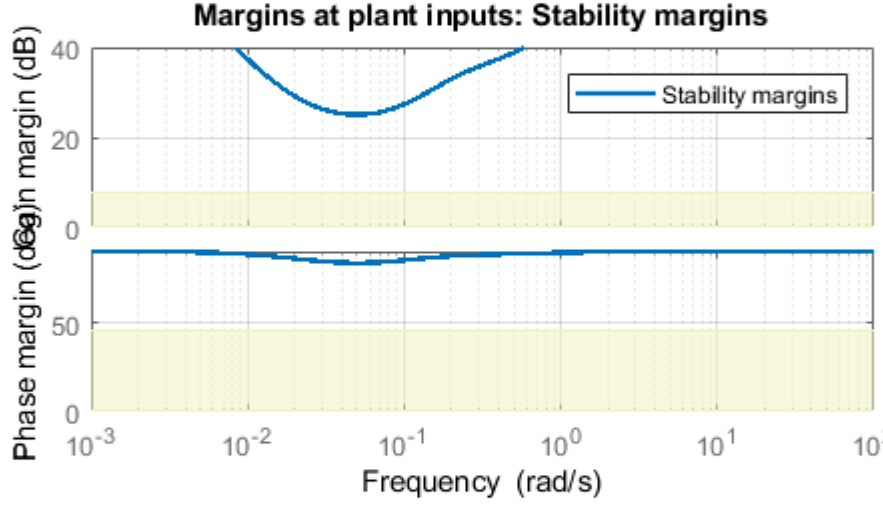


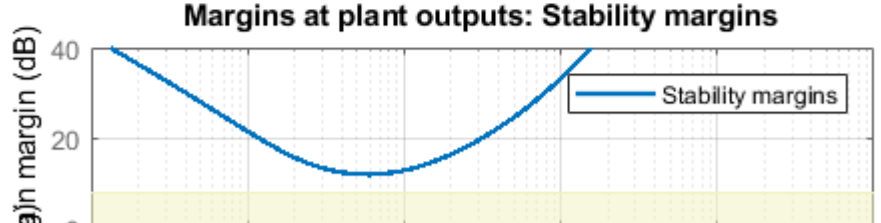
The decoupling matrix in the controller permits each channel of the two-channel output signal `y` to track the corresponding channel of the reference signal `r`, with minimal crosstalk. From the plot, you can how well this requirement is achieved when you tune the control system for bandwidth alone. If the crosstalk still exceeds your design requirements, you can use a `TuningGoal.Gain` requirement object to impose further restrictions on tuning.

Examine the frequency-domain response of the tuned result as an alternative method for validating the tuned controller.

```
figure('Position',[100,100,520,1000])
loopview(G,C,Info)
```


Open loop CG: Minimum and maximum loop gains (CrossTol = 0.5)
Singular Values (dB)
Frequency (rad/s)
S
T
Loop gain(s)
Scaled loop gains
Target loop shape


Margins at plant inputs: Stability margins
Phase margin (deg)
Frequency (rad/s)
Stability margins


Margins at plant outputs: Stability margins
Stability margins

The first plot shows that the open-loop gain crossovers fall within the specified interval `[0.1,1]`. This plot also includes the maximum and tuned values of the sensitivity function $S = (I - GC)^{-1}$ and complementary sensitivity $T = I - S$. The second and third plots show that the MIMO stability margins of the tuned system (blue curve) do not exceed the upper limit (yellow curve).

See Also

Related Examples

- "Decoupling Controller for a Distillation Column" (Control System Toolbox)

More About

- "Structure of Control System for Tuning With looptune" on page 11-2

Model Verification

Monitor Linear System Characteristics in Simulink Models

Simulink Control Design software provides Model Verification blocks to monitor time- and frequency-domain characteristics of a linear system on page 12-4 computed from a nonlinear Simulink model during simulation.

Use these blocks to:

- Verify that the linear system characteristics of any nonlinear Simulink model, including the following, remain within specified bounds during simulation:
  - Continuous- or discrete-time models
  - Multi-rate models
  - Models with time delays, represented using exact delay or Padé approximation
  - Discretized linear models computed from continuous-time models
  - Continuous-time models computed from discrete-time models
  - Resampled discrete-time models

  The linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO).
- View specified bounds and bound violations on linear analysis plots.

  Tip These blocks are same as the Linear Analysis Plots blocks except for different default settings of the bound parameters.
- Save the computed linear system to the MATLAB workspace.

The verification blocks assert when the linear system characteristic does not satisfy a specified bound, i.e., assertion fails. A warning message, reporting the assertion failure, appears at the MATLAB prompt. When assertion fails, you can:

- Stop the simulation and bring that block into focus.
- Evaluate a MATLAB expression.

You can use these blocks with the Simulink Model Verification blocks to design complex logic for assertion. For an example, see "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25.

You can use the Verification Manager tool in the Signal Builder to construct simulation tests for your model. For an example, see “Verifying Frequency-Domain Characteristics of an Aircraft”.

Note These blocks do not support code generation and can only be used in `Normal` simulation mode.

Define Linear System for Model Verification Blocks

To assert that the linear system characteristics satisfy specified bounds, the Model Verification blocks compute a linear system from a nonlinear Simulink model.

For the software to compute a linear system, you must specify:

- Linearization inputs and outputs on page 14-5

  Linearization inputs and outputs define the portion of the model to linearize. A linearization input defines an input while a linearization output defines an output of the linearized model. To compute a MIMO linear system, specify multiple inputs and outputs.
- When to compute the linear system on page 14-10

  You can compute the linear system and assert bounds at:

  - Default simulation snapshot time. Typically, simulation snapshots are the times when your model reaches steady state.
  - Multiple simulation snapshots.
  - Trigger-based simulation events

For more information, see the following examples:

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15

Verifiable Linear System Characteristics

The following table summarizes the linear system characteristics you can specify bounds on and assert that the bounds are satisfied during simulation.

Block	Plot Type	Bounds on…
Check Bode Characteristics	Bode	Upper and lower Bode magnitude
Check Gain and Phase Margins	• Bode • Nichols • Nyquist • Table	Gain and phase margins
Check Nichols Characteristics	Nichols	• Open-loop gain and phase • Closed-loop peak gain
Check Pole-Zero Characteristics	Pole-Zero	Approximate second-order characteristics, such as settling time, percent overshoot, damping ratio and natural frequency, on the pole locations
Check Singular Value Characteristics	Singular Value	Upper and lower singular values
Check Linear Step Response Characteristics	Step Response	Step response characteristics

Specify the bounds in the **Bounds** tab of the block's Block Parameters dialog box or programmatically. For more information, see the corresponding block reference pages.

Verify Model at Default Simulation Snapshot Time

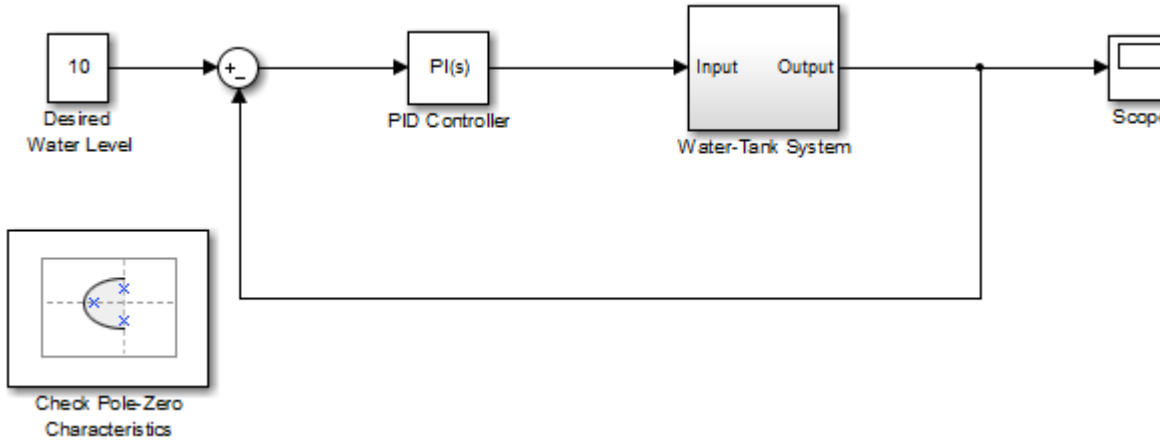
This example shows how to assert that bounds on the linear system characteristics of a nonlinear Simulink model, computed at the default simulation snapshot time of 0, are satisfied during simulation.

1. Open a nonlinear Simulink model. For example:

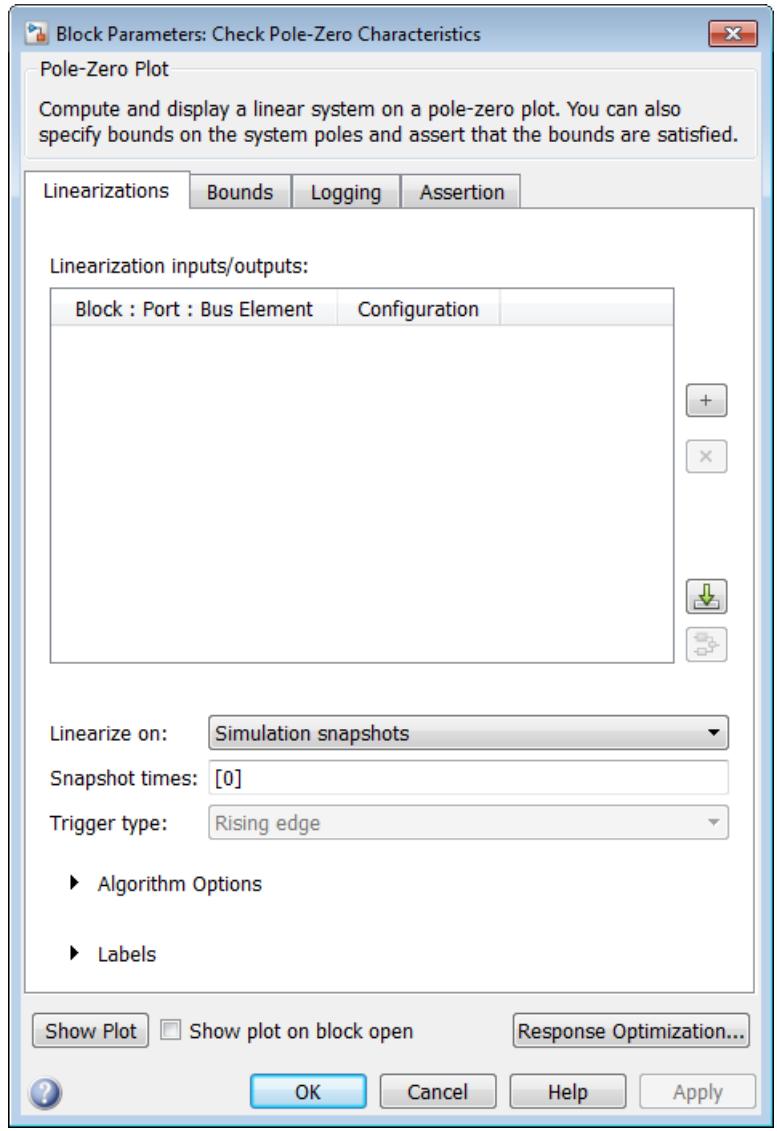
   ```
   watertank
   ```

2. Open the Simulink Library Browser by selecting **View > Library Browser** in the Simulink Editor.
3. Add a model verification block to the Simulink model.
   a. In the **Simulink Control Design** library, select **Model Verification**.
   b. Drag and drop a block, such as the Check Pole-Zero Characteristics block, into the Simulink Editor.

   The model now resembles the following figure.



4. Double-click the block to open the Block Parameters dialog box.



To learn more about the block parameters, see the block reference pages.

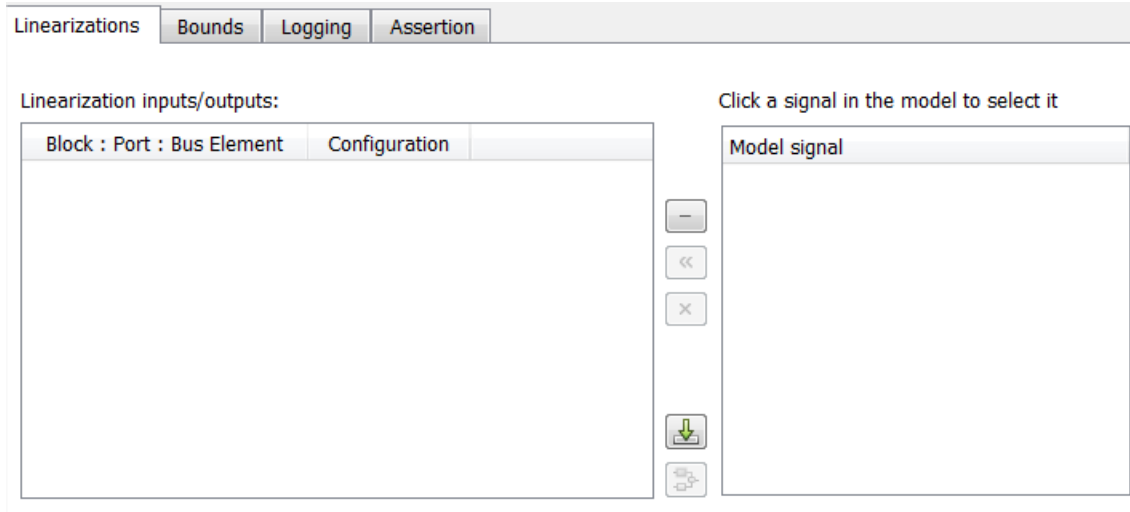
5 Specify the linearization input and output to compute the closed-loop poles and zeros.

Tip If you have defined the linearization input and output in your Simulink model, click  to automatically populate the **Linearization inputs/outputs** table with I/Os from the model.

a To specify an input:

i Click  adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.

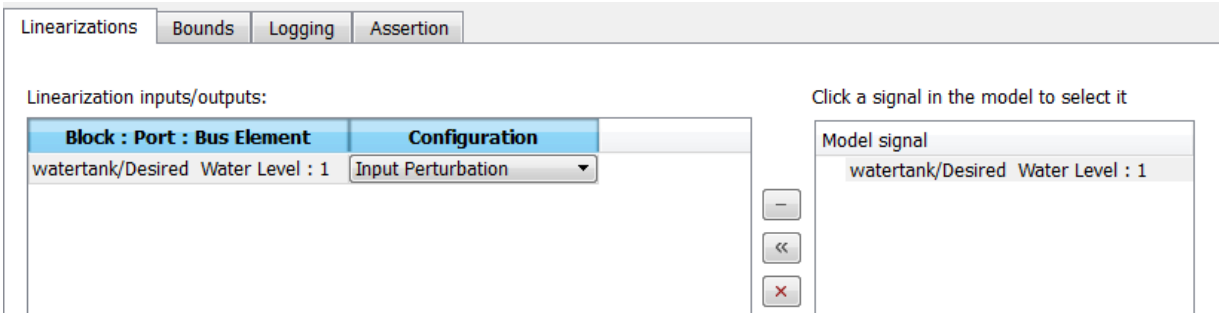


ii In the Simulink model, click the output signal of the `Desired Water Level` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.

Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

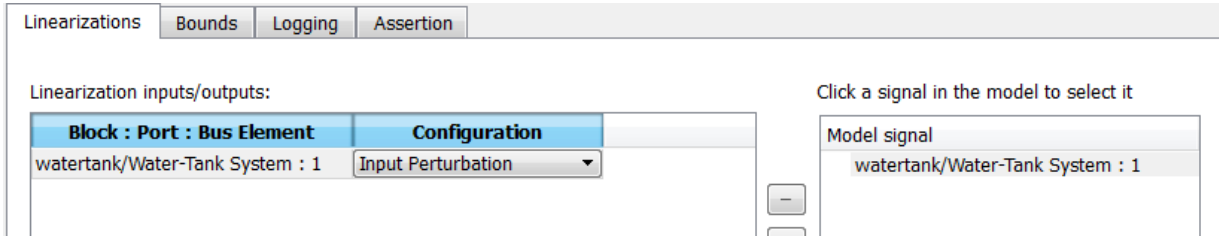
iii Click « to add the signal to the **Linearization inputs/outputs** table.



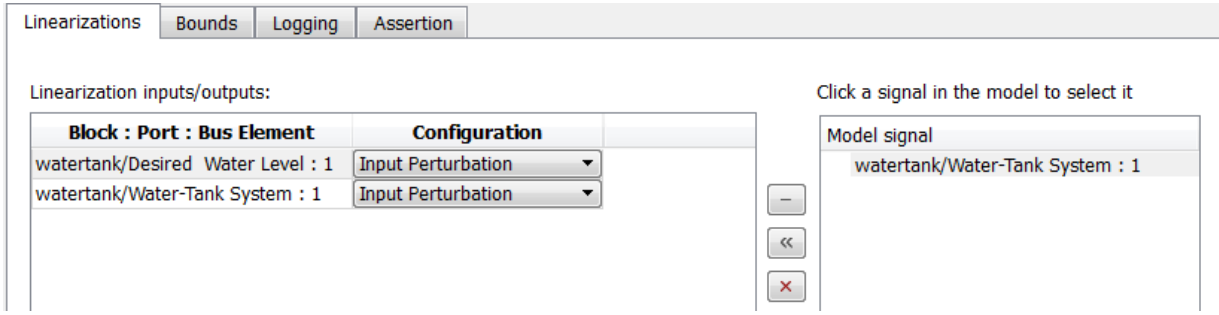
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



ii Click « to add the signal to the **Linearization inputs/outputs** table.



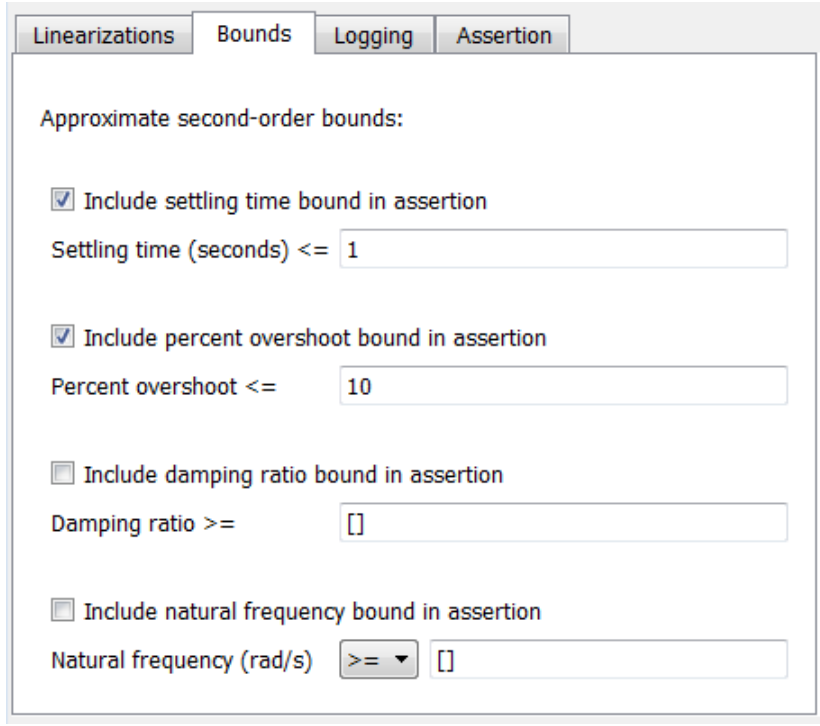
Note To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Output Measurement` for **watertank/Water-Tank System: 1**.

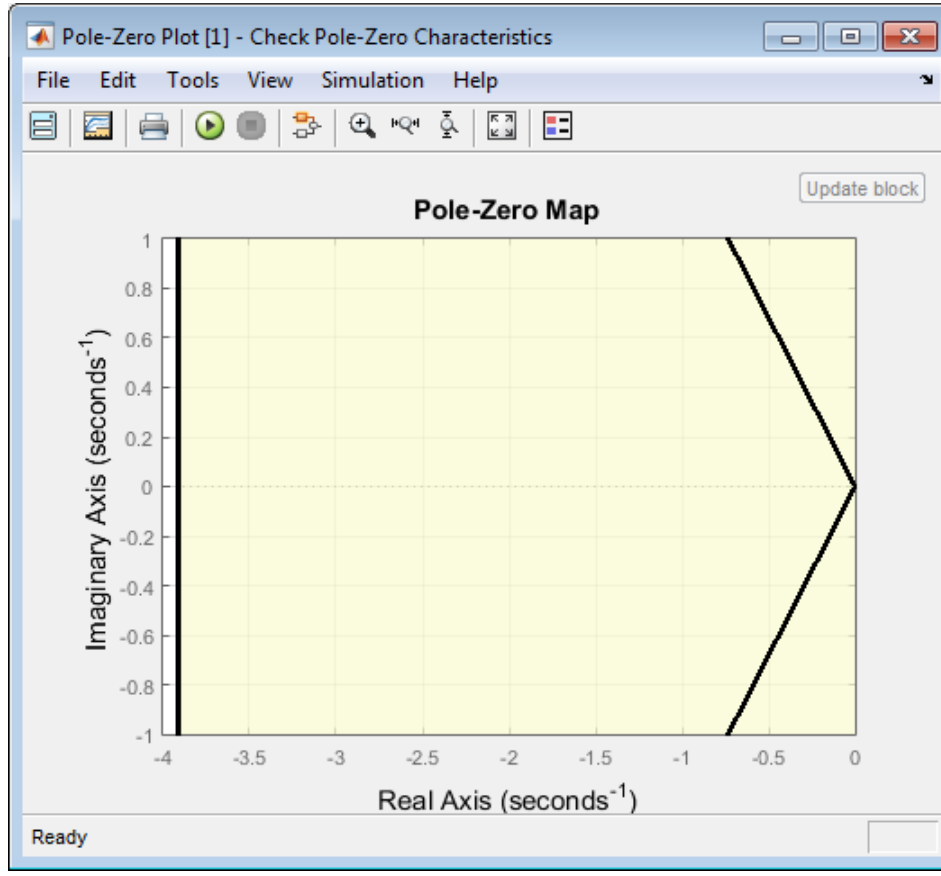
Note The I/Os include the feedback loop in the Simulink model. The software computes the poles and zeros of the closed-loop system.

iv Click  to collapse the **Click a signal in the model to select it** area.

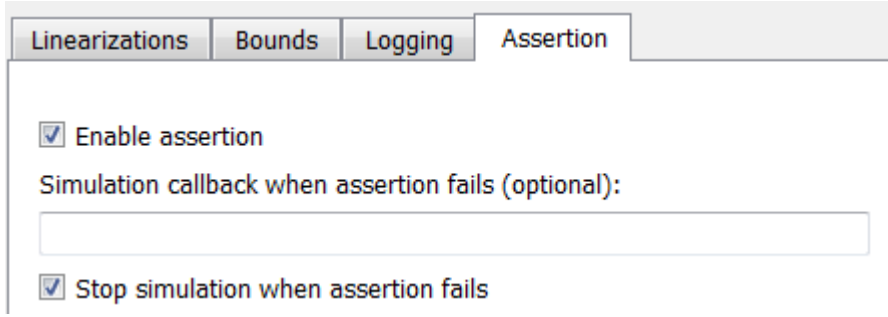
6 Specify bounds for assertion. In this example, you use the default approximate second-order bounds, specified in **Bounds** tab of the Block Parameters dialog box.



View the bounds on the pole-zero map by clicking **Show Plot** to open a plot window.



7 Stop the simulation if assertion fails by selecting **Stop simulation when assertion fails** in the **Assertion** tab.

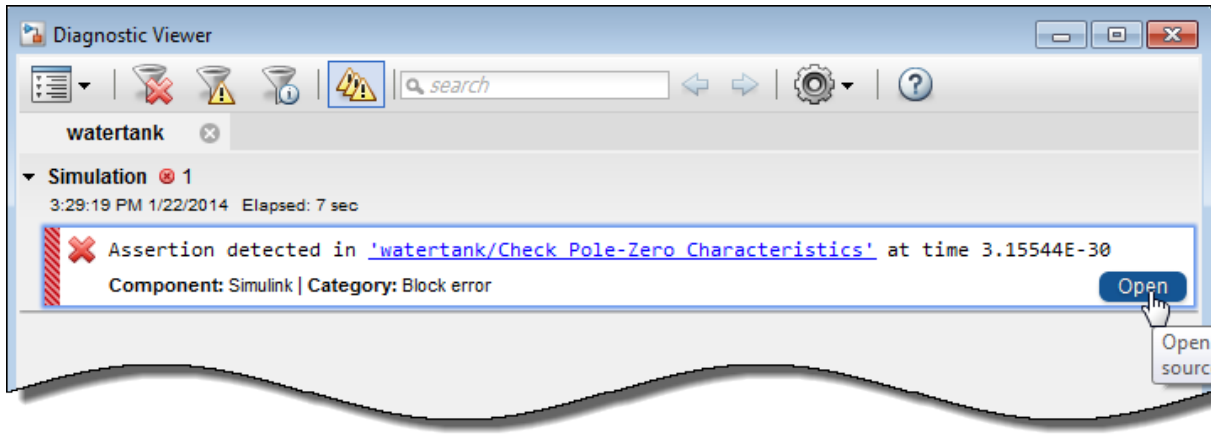


8 Click **Apply** to apply all changed settings to the block.

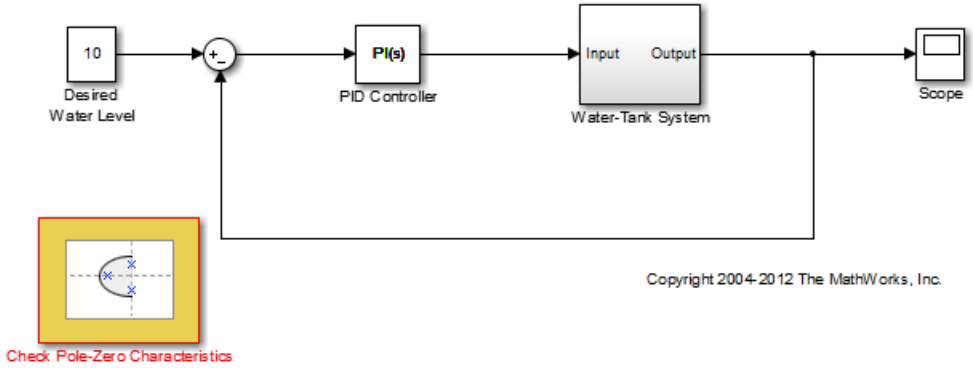
9 Simulate the model by clicking  in the plot window.

Alternatively, you can simulate the model from the Simulink Editor.

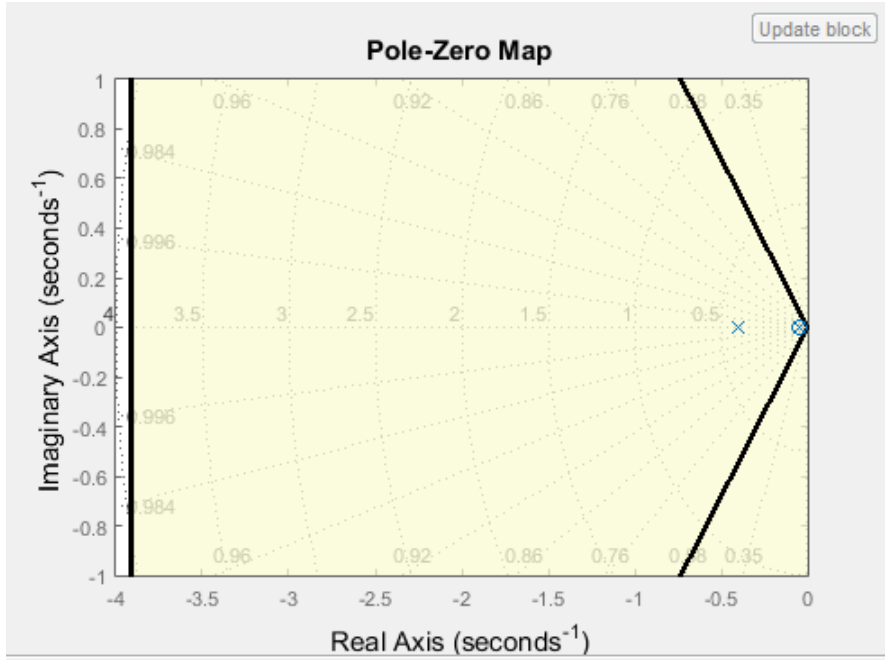
The software linearizes the portion of the model between the linearization input and output at the default simulation time of 0, specified in **Snapshot times** block parameter. When the software detects that a pole violates a specified bound, the simulation stops. The Diagnostics Viewer opens reporting the block that asserts.



Click **Open** to highlight the block that asserts in the Simulink model.



The closed-loop pole and zero locations of the computed linear system appear as x and o markings in the plot window. You can also view the bound violation in the plot.



Verify Model at Multiple Simulation Snapshots

This example shows how to:

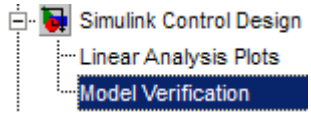
- Add multiple bounds.
- Check that the linear system characteristics of a nonlinear Simulink model satisfy the bounds at multiple simulation snapshots
- Modify bounds graphically
- Disable bounds during simulation

1. Open a nonlinear Simulink model. For example:

   ```
   watertank
   ```

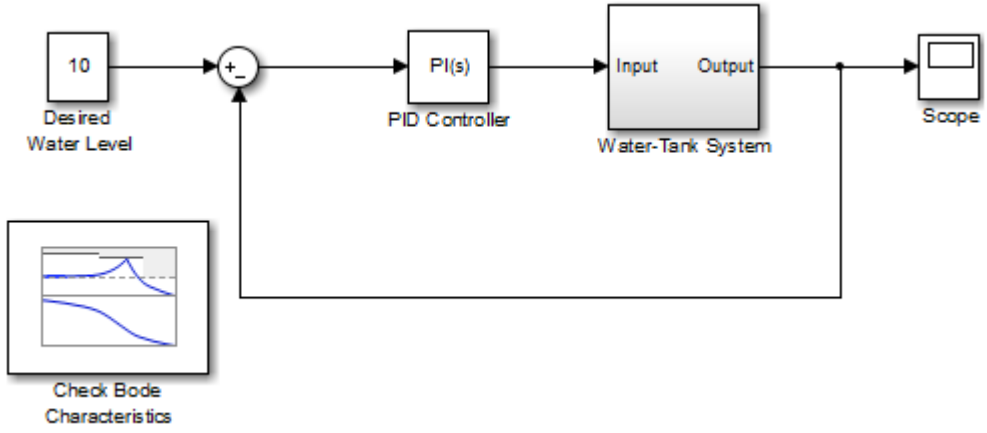
2. Open the Simulink Library Browser by selecting **View** > **Library Browser** in the Simulink Editor.
3. Add a model verification block to the Simulink model.

   a. In the **Simulink Control Design** library, select **Model Verification**.

   b. Drag and drop a block, such as the Check Bode Characteristics block, into the Simulink Editor.

   The model now resembles the following figure.



4 Double-click the block to open the Block Parameters dialog box.

To learn more about the block parameters, see the block reference pages.

5 Specify the linearization I/O points.

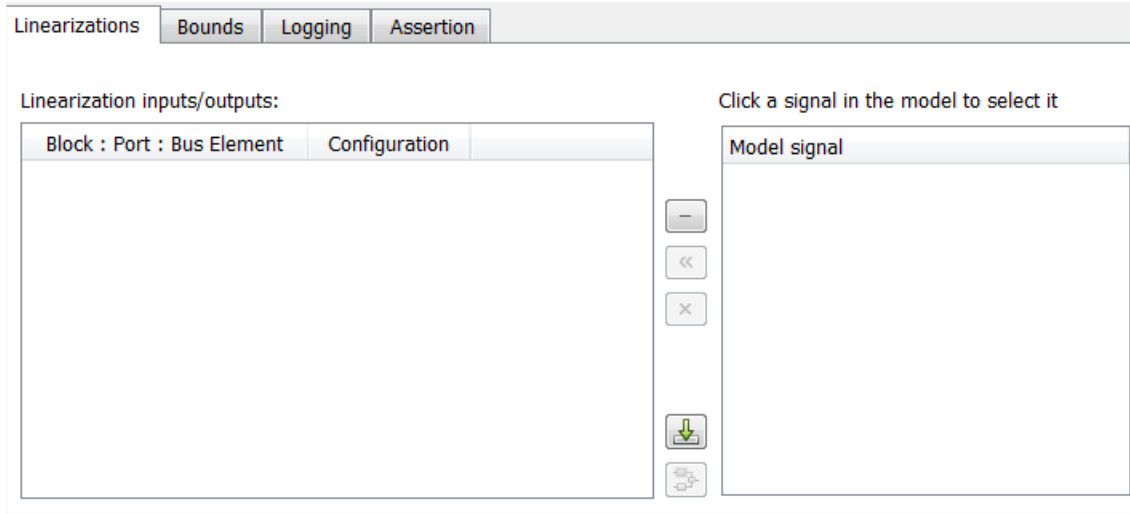
The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

a To specify an input:

i Click  adjacent to the **Linearization inputs/outputs** table.

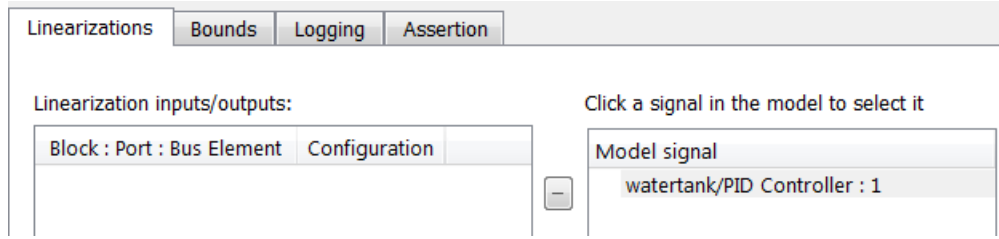
The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.



Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

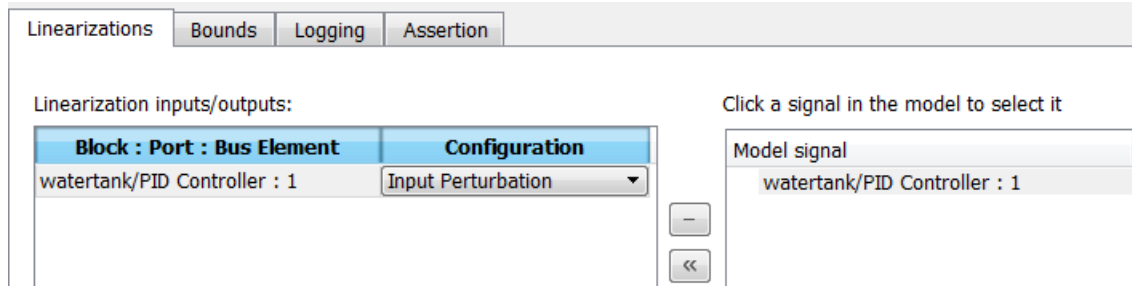
ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



iii Click « to add the signal to the **Linearization inputs/outputs** table.

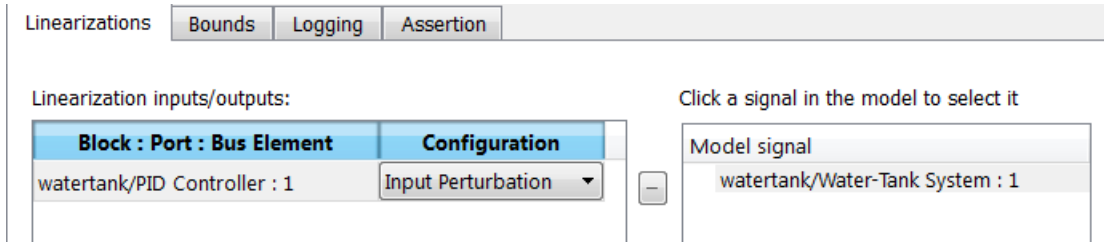
To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.



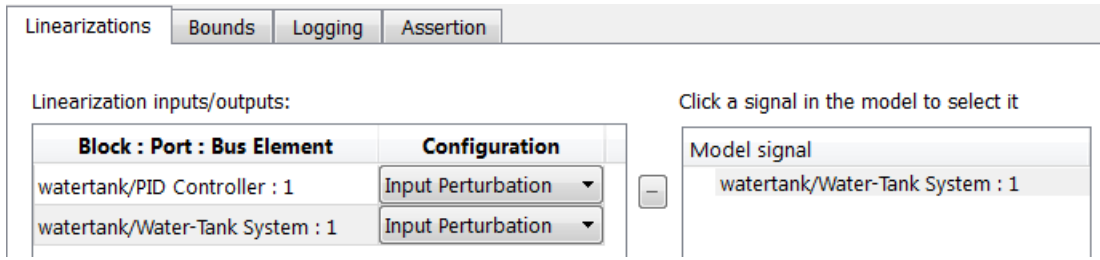
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



ii Click « to add the signal to the **Linearization inputs/outputs** table.

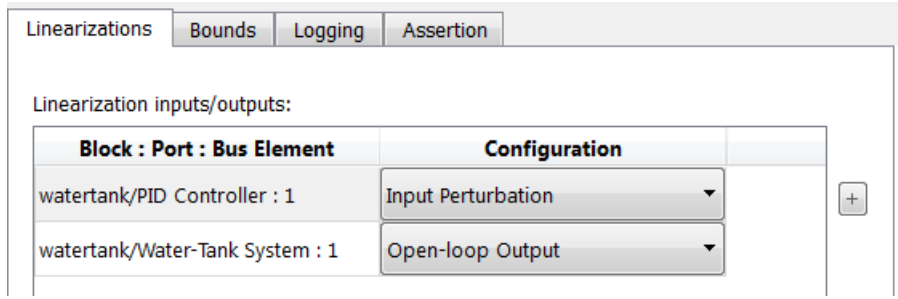


To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Note To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



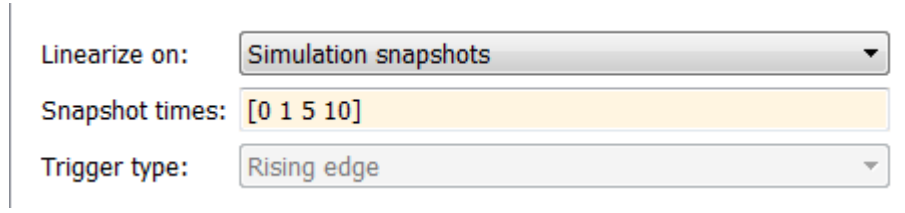
c Click − to collapse the **Click a signal in the model to select it** area.

Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear in the **Linearization inputs/outputs** table.

6 Specify simulation snapshot times.

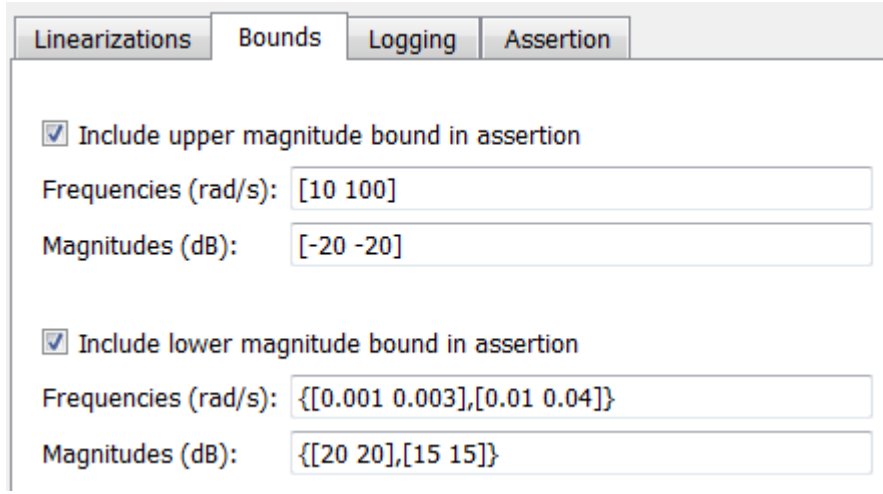
a In the **Linearizations** tab, verify that `Simulation snapshots` is selected in **Linearize on**.

b In the **Snapshot times** field, type `[0 1 5 10]`.



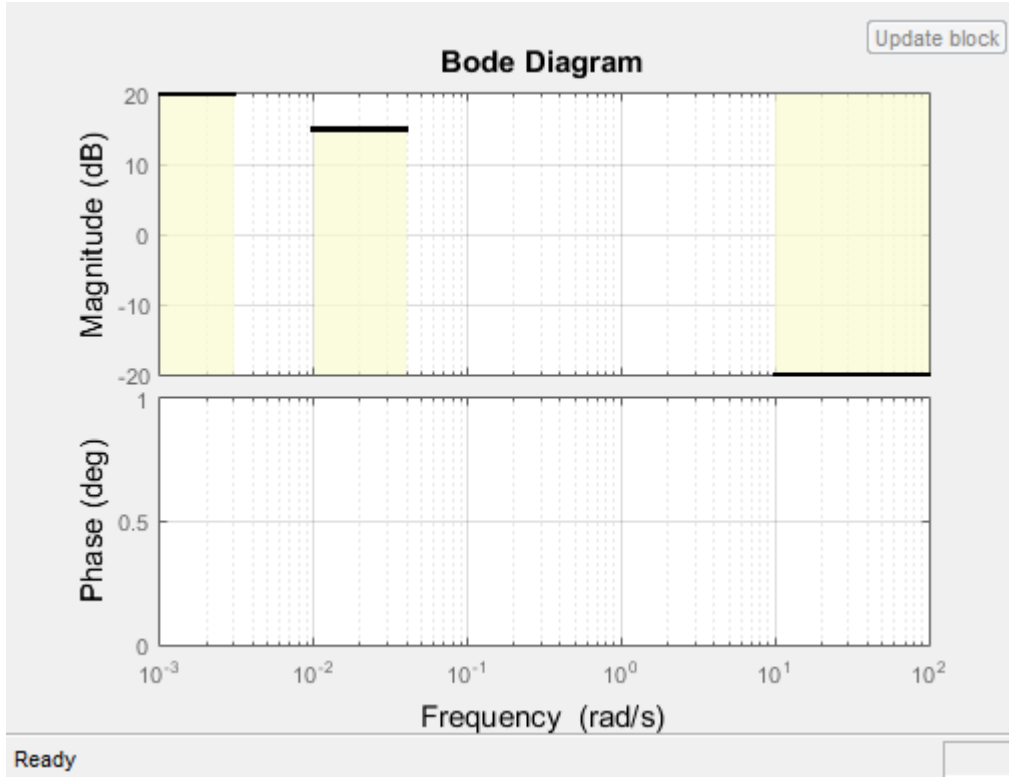
7 Specify multiple bound segments for assertion in the **Bounds** tab of the Block Parameters dialog box. In this example, enter the following lower magnitude bounds:

- **Frequencies (rad/s)** — `{[0.001 0.003],[0.01 0.04]}`
- **Magnitudes (dB)** — `{[20 20],[15 15]}`



Click **Apply** to apply the parameter changes to the block.

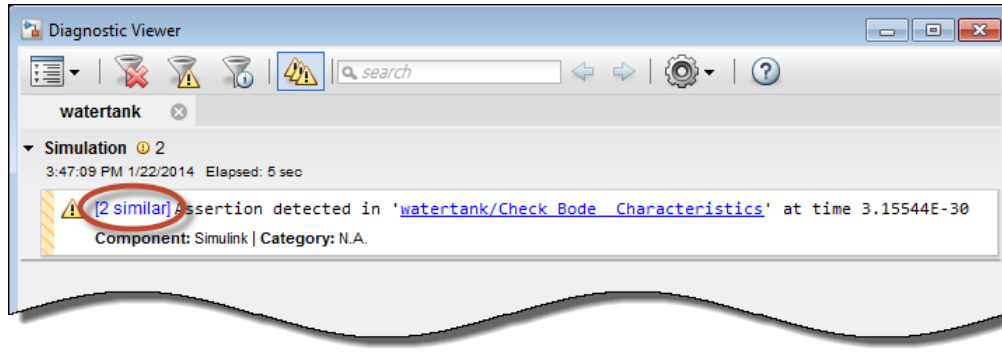
Click **Show Plot** to view the bounds on the Bode magnitude plot.



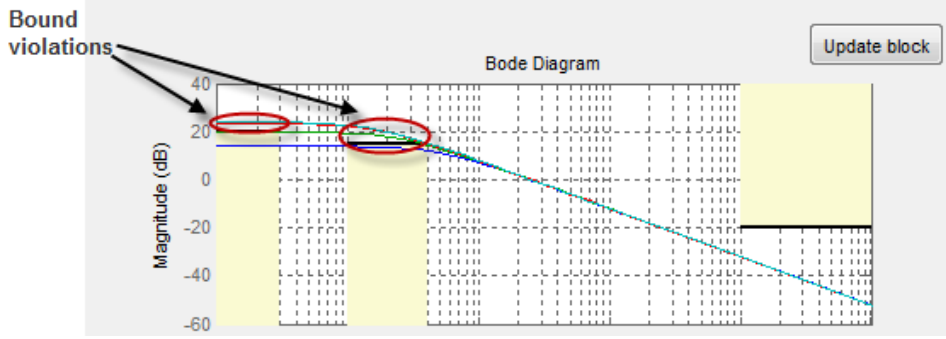
8 Simulate the model by clicking  in the plot window.

Alternatively, you can simulate the model from the Simulink Editor.

The software linearizes the portion of the model between the linearization input and output at the simulation times of 0,1, 5 and 10. When the software detects that the linear system computed at times 0 and 1 violate a specified lower magnitude bound, warning messages appear in the Diagnostic Viewer window. Click the link at the bottom of the Simulink model to open this window. Click the link in the window to view the details of the assertion.

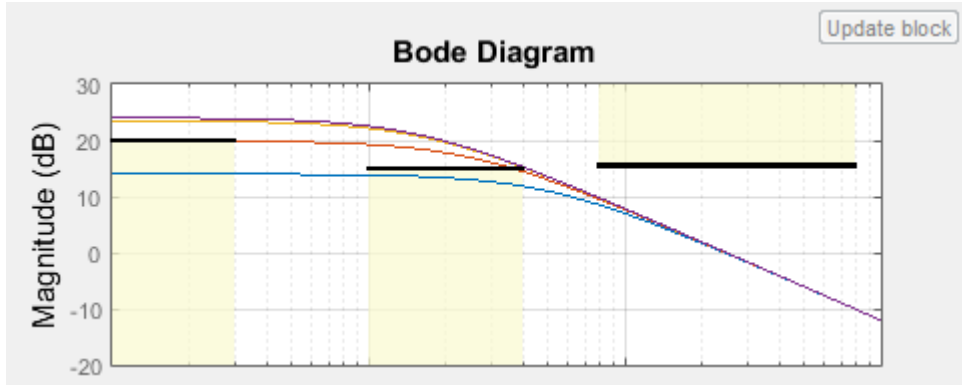


You can also view the bound violations on the plot window.

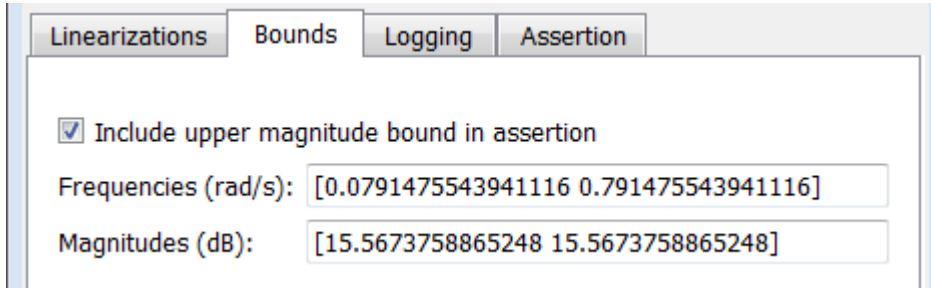


9 Modify a bound graphically. For example, to modify the upper magnitude bound graphically:

a In the plot window, click the bound segment to select it and then drag it to the desired location.

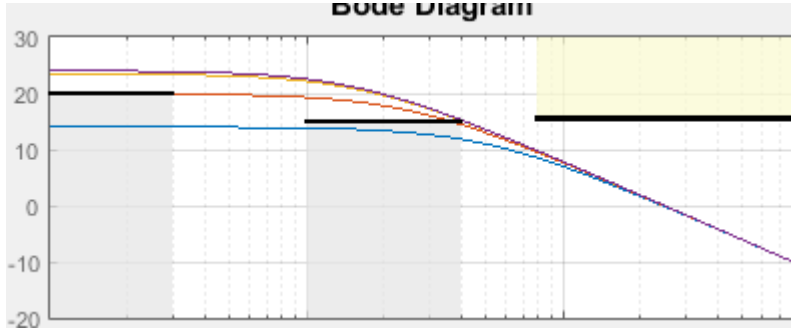


b Click **Update block** to update the new values in the Bounds tab of the Block Parameters dialog box.



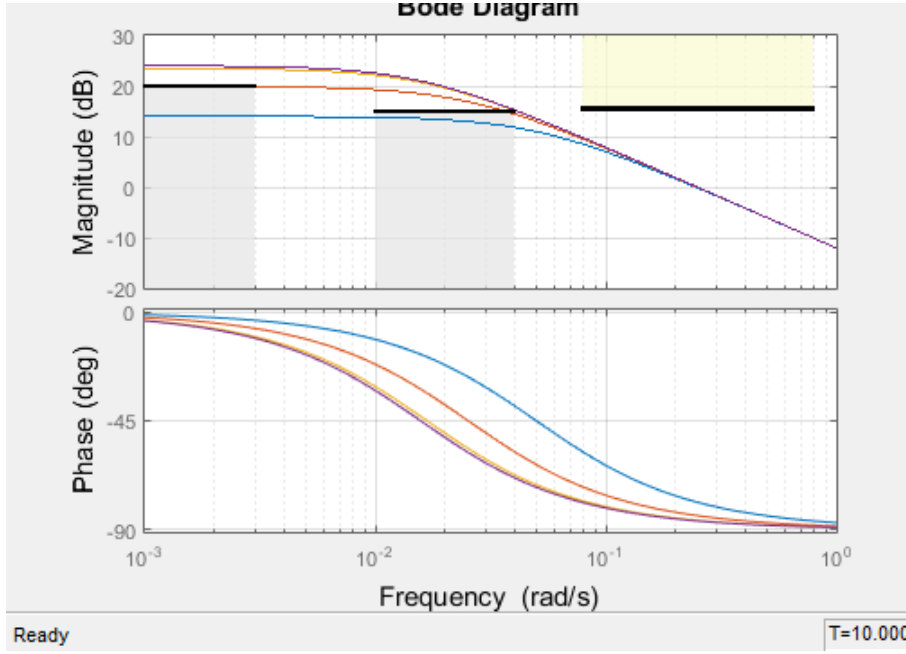
10 Disable the lower bounds to exclude them from asserting. Clear the **Include lower magnitude bounds in assertion** option in the Block Parameters dialog box. Then, click **Apply**.

The lower bounds are now grey-out in the plot window, indicating that they are excluded from assertion.



11 Resimulate the model to check if bounds are satisfied.

The software satisfies the specified upper magnitude bound, and therefore the software no longer reports an assertion failure.



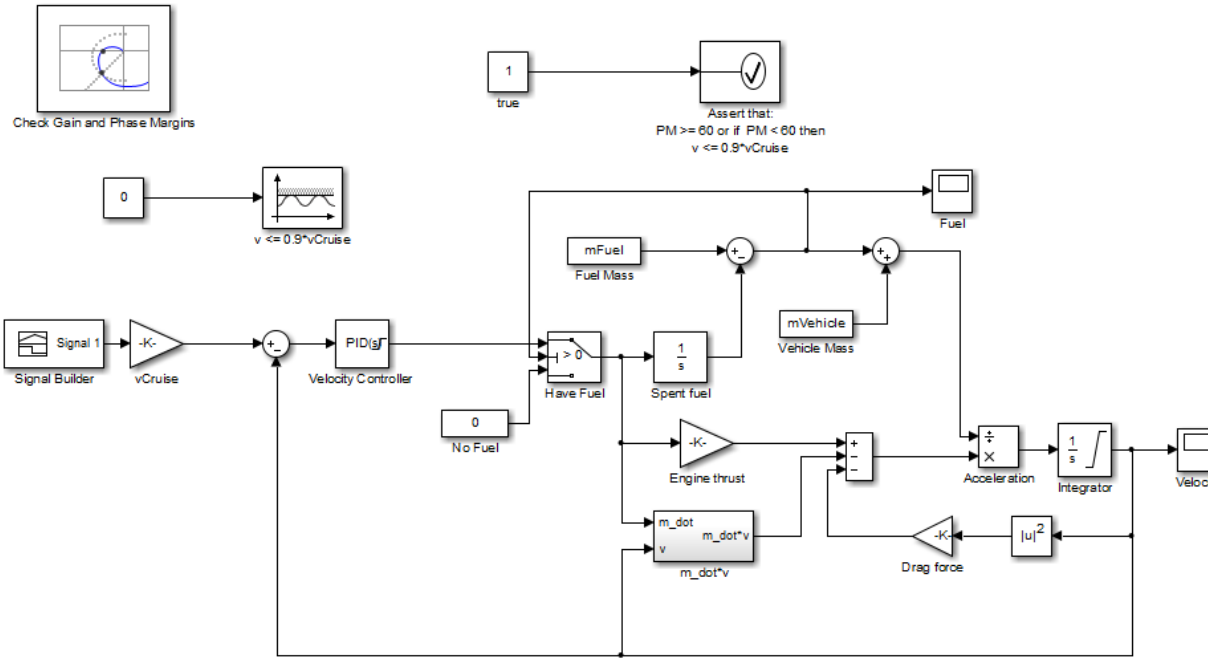
Verify Model Using Simulink Control Design and Simulink Verification Blocks

This example shows how to use a combination of Simulink Control Design and Simulink verification blocks, to assert that the linear system characteristics satisfy one of the following bounds:

- Phase margin greater than 60 degrees
- Phase margin less than 60 degrees and the velocity less than or equal to 90% of the cruise velocity.

1 Open the Simulink model of an aircraft.

```
scdmultiplechecks
```



The aircraft model is based on a long-haul passenger aircraft flying at cruising altitude and speed. The aircraft starts with a full fuel load and follows a pre-

specified 8-hour velocity profile. The model is a simplified version of a velocity control loop, which adjusts the fuel flow rate to control the aircraft velocity.

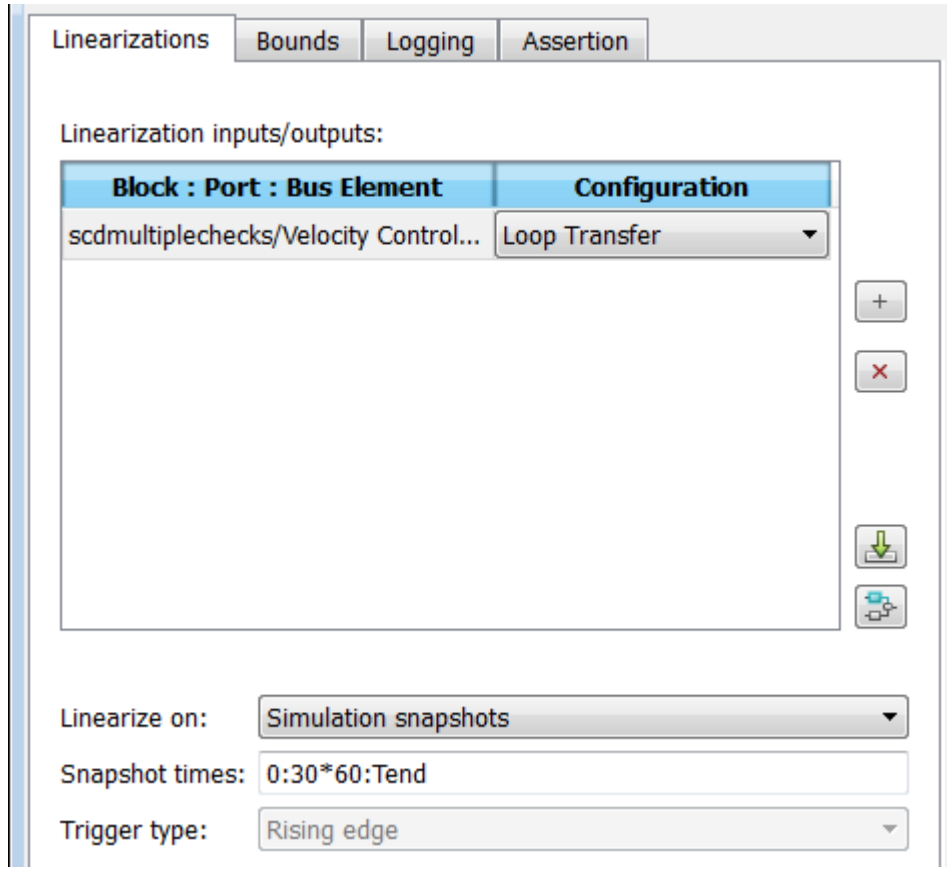
The model includes blocks to model:

- Fuel consumption and resulting changes in aircraft mass
- Nonlinear draft effects limiting aircraft velocity

Constants used in the model, such as the drag coefficient, are defined in the model workspace and initialized from a script.

The `v <= 0.9*vCruise` and `Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise` blocks are Check Static Upper Bound and Assertion blocks, respectively, from the Simulink Model Verification library. In this example, you use these blocks with the Check Gain and Phase Margins block to design a complex logic for assertion.

2 View the linearization input, output and settings in the **Linearizations** tab of the Check Gain and Phase Margins block parameters dialog box.

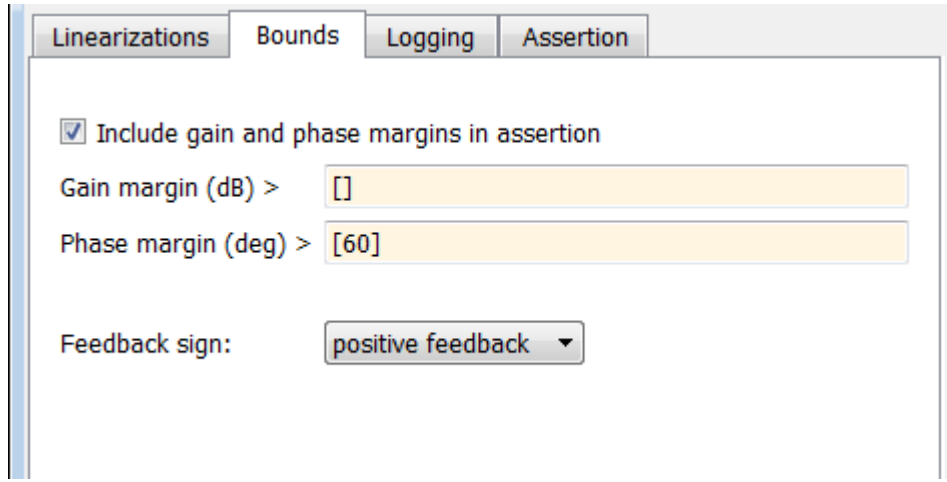


The model has already been configured with:

- Linearization input and output for computing gain and phase margins
- Settings to compute the linear system

The software linearizes the loop seen by the Velocity Controller block every 30 minutes of simulated time and computes the gain and phase margins.

3 Specify phase margin bounds in the **Bounds** tab of the Check Gain and Phase Margins block.



In this example, the linearization input and output include the summation block with negative feedback. Change the **Feedback sign**, used to compute the margin, to `positive feedback`.

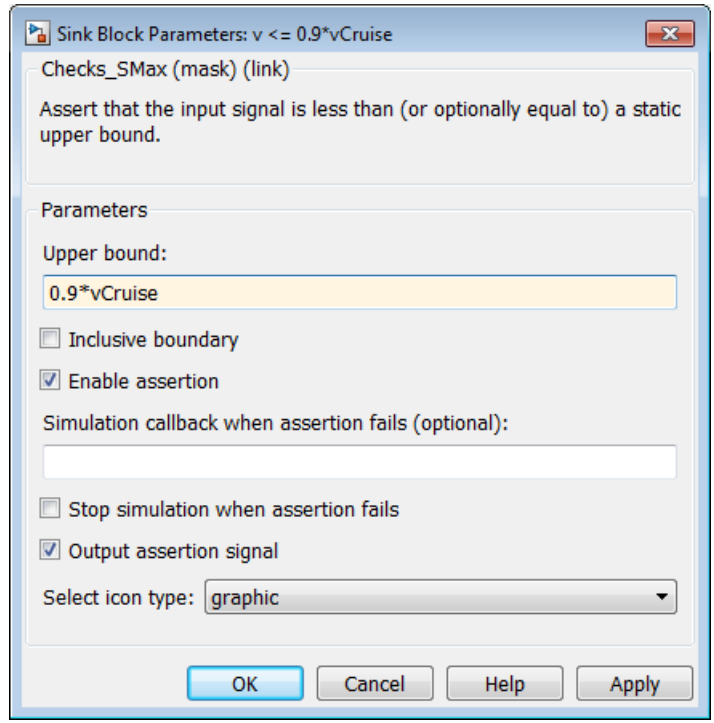
To view the phase margins to be computed later during simulation, specify `Tabular` in **Plot type**, and click **Show Plot**.

4 Design assertion logic that causes the verification blocks to assert when the phase margin is greater than 60 degrees or if the phase margin is less than 60 degrees, the velocity is less than or equal to 90% the cruise velocity.

a In the Check Gain and Phase Margins Block Parameters dialog box, in the **Assertion** tab, select **Output assertion signal**, and click **Apply**.

This action adds an output port z^{-1} to the block.

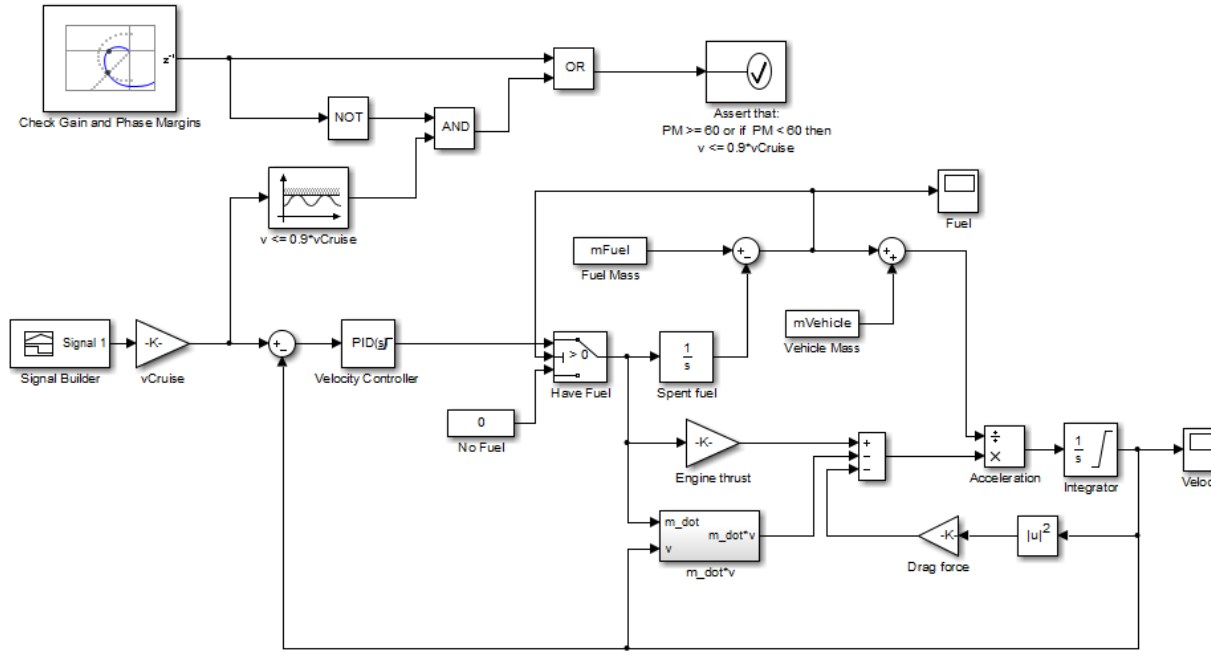
b Double-click the `v <= 0.9*vCruise` block, and specify the block parameters, as shown in the following figure. After setting the parameters, click **Apply**.



These parameters configure the block to:

- Check if the aircraft velocity exceeds the cruise velocity by 0.9 times
- Add an output port to the block

c Connect the Check Gain and Phase Margins, `v <= 0.9*vCruise` and `Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise` blocks, as shown in the following figure.



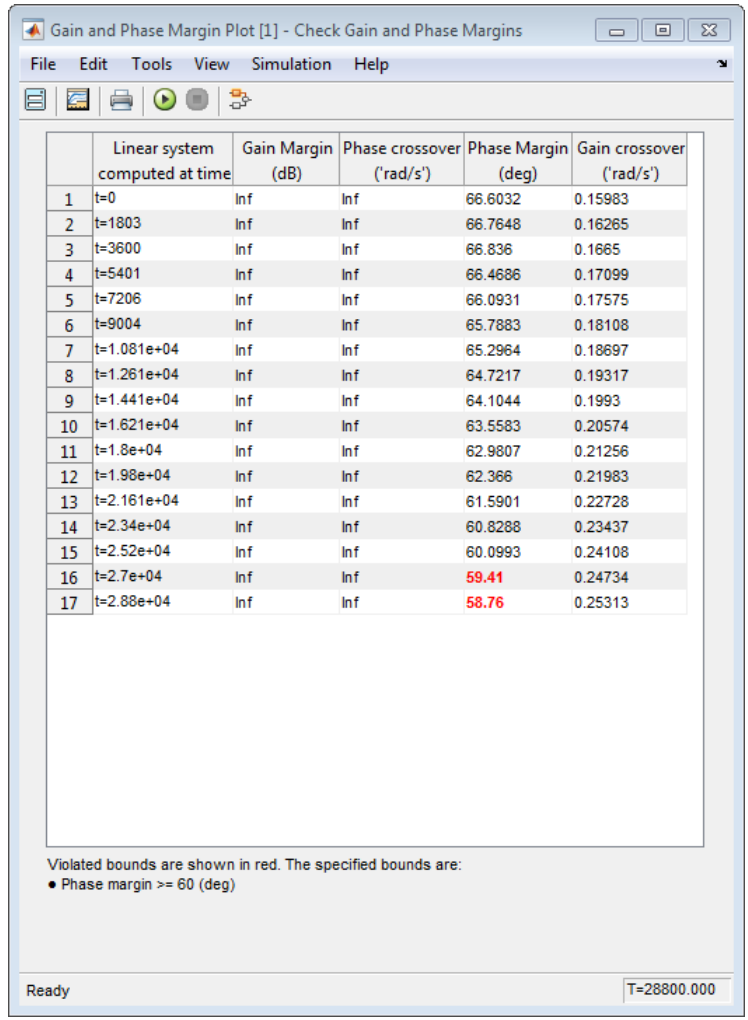
This connection causes the `Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise` block to assert and stop the simulation if the phase margin is less than 60 degrees and the velocity is greater than 90% of the cruise velocity.

Alternatively, you can type `scdmultiplechecks_final` at the MATLAB prompt to open a Simulink model already configured with these settings.

5 Simulate the model by selecting **Simulation** > **Run** in the Simulink Editor.

During simulation:

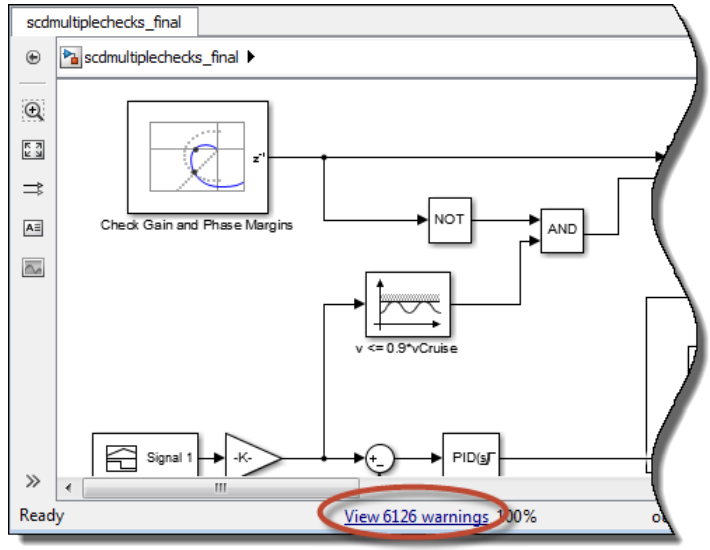
- The `v <= 0.9*vCruise` block asserts multiple times.
- The Check Gain and Phase Margins block asserts two times. You can view the phase margins that violate the bound in the plot window.



	Linear system computed at time	Gain Margin (dB)	Phase crossover ('rad/s')	Phase Margin (deg)	Gain crossover ('rad/s')
1	t=0	Inf	Inf	66.6032	0.15983
2	t=1803	Inf	Inf	66.7648	0.16265
3	t=3600	Inf	Inf	66.836	0.1665
4	t=5401	Inf	Inf	66.4686	0.17099
5	t=7206	Inf	Inf	66.0931	0.17575
6	t=9004	Inf	Inf	65.7883	0.18108
7	t=1.081e+04	Inf	Inf	65.2964	0.18697
8	t=1.261e+04	Inf	Inf	64.7217	0.19317
9	t=1.441e+04	Inf	Inf	64.1044	0.1993
10	t=1.621e+04	Inf	Inf	63.5583	0.20574
11	t=1.8e+04	Inf	Inf	62.9807	0.21256
12	t=1.98e+04	Inf	Inf	62.366	0.21983
13	t=2.161e+04	Inf	Inf	61.5901	0.22728
14	t=2.34e+04	Inf	Inf	60.8288	0.23437
15	t=2.52e+04	Inf	Inf	60.0993	0.24108
16	t=2.7e+04	Inf	Inf	59.41	0.24734
17	t=2.88e+04	Inf	Inf	58.76	0.25313

- The `Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise` does not encounter the assertion condition. Therefore, the simulation does not stop.

6 Click the link at the bottom of the Simulink model to open the Diagnostic Viewer window.



When a block asserts, warnings appear in this window. You can view the details of the assertions by clicking the link in this window.

Alphabetical List

Linear Analysis Tool

Linearize Simulink models

Description

The **Linear Analysis Tool** lets you perform linear analysis of nonlinear Simulink models.

Using this tool you can:

- Interactively linearize models at different operating points.
- Interactively obtain operating points by trimming or simulating models.
- Perform exact linearization of nonlinear models.
- Perform frequency response estimation of nonlinear models.
- Batch linearize models for varying parameter values.
- Generate MATLAB code for performing linearization tasks.
- Generate MATLAB code for computing operating points.

Open the Linear Analysis Tool App

- Simulink model editor: Select **Analysis > Control Design > Linear Analysis**.
- Simulink model editor: Select **Analysis > Control Design > Frequency Response Estimation**.
- Simulink model editor: Right-click a block, and select **Linear Analysis > Linearize Block**.

Examples

- “Linearize Simulink Model at Model Operating Point” on page 2-69
- “Linearize at Trimmed Operating Point” on page 2-85
- “Linearize at Simulation Snapshot” on page 2-91

- "Estimate Frequency Response Using Linear Analysis Tool" on page 5-26
- "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29
- "Analyze Results Using Linear Analysis Tool Response Plots" on page 2-146
- "Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75

Parameters

Linear Analysis Tab

`Analysis I/Os` — Linearization inputs, outputs, and loop openings

`Model I/Os` (default) | linearization I/O set

Linearization inputs, outputs, and loop openings. The currently active I/O set is displayed. To change the I/O set, select one of the following:

- `Model I/Os` — Use the inputs, outputs, and loop openings specified in the Simulink model. For more information on specifying analysis points in your model, see "Specify Portion of Model to Linearize in Simulink Model" on page 2-21.
- `Root Level Inports and Outports` — Use the root level inputs and outputs of the Simulink model.
- `Linearize the Currently Selected Block` — Use the input and output ports of the currently selected block in the Simulink model.
- `Create New Linearization I/Os` — Specify inputs, outputs, and loop openings. For more information, see "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29.
- `Existing I/Os` — Select a previously created I/O set.
- `View/Edit` — View or edit the currently selected operating point. For more information, see "Edit Analysis Points" on page 2-34.

`Operating Point` — Linearization operating point

`Model Initial Condition` (default) | operating point

Linearization operating point. The current operating point is displayed. To change the operating point, select one of the following:

- `Model Initial Condition` — Use the initial conditions defined in the Simulink.

- `Linearize At` — Simulate the model using the model initial conditions, and use the simulation snapshot at the specified time as the operating point. For more information, see "Linearize at Simulation Snapshot" on page 2-91.
- `Linearize at Multiple Points` — Select multiple previously created operating points.
- `Existing Operating points` — Select a previously created operating point.
- `Trim Model` — Compute a steady-state operating point. For more information, see "Compute Steady-State Operating Point from State Specifications" on page 1-14 and "Compute Steady-State Operating Point from Output Specifications" on page 1-28.
- `Take Simulation Snapshot` — Simulate the model using the model initial conditions, and compute an operating point at the specified simulation snapshot times. For more information, see "Compute Operating Points at Simulation Snapshots" on page 1-78.
- `View/Edit` — View or edit the currently selected operating point.

`Parameter Variations` — Parameters to vary for batch linearization

`None` (default) | parameters to vary

To vary parameters for batch linearization, in the drop-down list, click `Select parameters to vary`. On the **Parameter Variations** tab, specify the parameters to vary.

For more information, see "Specify Parameter Samples for Batch Linearization" on page 3-62.

`Result Viewer` — Open linearization result viewer

cleared (default) | checked

Select to display result details after linearization. For more information, see "View Linearized Model Equations Using Linear Analysis Tool" on page 2-144.

`Linearization Advisor` — Collect diagnostic information and open Linearization Advisor

cleared (default) | checked

Select to collect diagnostic information during linearization and open an **Advisor** tab for interactive troubleshooting of linearization problems. For more information, see "Troubleshoot Linearization Results in Linear Analysis Tool" on page 4-21.

Note The **Linear Analysis Tool** only collects diagnostic information when **Linearization Advisor** is checked before performing a linearization task.

Estimation Tab

`Input Signal` — Estimation input signal

`Sinestream` | `Fixed Sample Time Sinestream` | `Chirp` | `Random`

Estimation input signal. The current input signal is displayed. To change the input signal, select one of the following:

- `Sinestream` — Create an input signal that consists of adjacent sine waves of varying frequencies. For more information, see "Create Sinestream Input Signals" on page 5-13.
- `Fixed Sample Time Sinestream` — Create a discrete-time sinestream input with a specified sample time.
- `Chirp` — Create a swept-frequency cosine input signal. For more information, see "Create Chirp Input Signals" on page 5-19.
- `Random` — Create a random input signal.

`Analysis I/Os` — Linearization inputs, outputs, and loop openings

`Model I/Os` (default) | linearization I/O set

Linearization inputs, outputs, and loop openings. The currently active I/O set is displayed. To change the I/O set, select one of the following:

- `Model I/Os` — Use the inputs, outputs, and loop openings specified in the Simulink model. For more information on specifying analysis points in your model, see "Specify Portion of Model to Linearize in Simulink Model" on page 2-21.
- `Root Level Inports and Outports` — Use the root level inputs and outputs of the Simulink model.
- `Linearize the Currently Selected Block` — Use the input and output ports of the currently selected block in the Simulink model.
- `Create New Linearization I/Os` — Specify inputs, outputs, and loop openings. For more information, see "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29.
- `Existing I/Os` — Select a previously created I/O set.

- `View/Edit` — View or edit the currently selected operating point. For more information, see “Edit Analysis Points” on page 2-34.

`Operating Point` — Linearization operating point
`Model Initial Condition` (default) | operating point

Linearization operating point. The current operating point is displayed. To change the operating point, select one of the following:

- `Model Initial Condition` — Use the initial conditions defined in the Simulink.
- `Linearize At` — Simulate the model using the model initial conditions, and use the simulation snapshot at the specified time as the operating point. For more information, see “Linearize at Simulation Snapshot” on page 2-91.
- `Linearize at Multiple Points` — Select multiple previously created operating points.
- `Existing Operating points` — Select a previously created operating point.
- `Trim Model` — Compute a steady-state operating point. For more information, see “Compute Steady-State Operating Point from State Specifications” on page 1-14 and “Compute Steady-State Operating Point from Output Specifications” on page 1-28.
- `Take Simulation Snapshot` — Simulate the model using the model initial conditions, and compute an operating point at the specified simulation snapshot times. For more information, see “Compute Operating Points at Simulation Snapshots” on page 1-78.
- `View/Edit` — View or edit the currently selected operating point.

`Result Viewer` — Open estimation result viewer
cleared (default) | checked

Select to display result details about the estimation configuration and input signal used for estimation.

`Diagnostic Viewer` — Collect diagnostic information and open diagnostic viewer
cleared (default) | checked

Select to collect diagnostic information that displays after estimation. You can use the diagnostic information to analyze the estimation result and troubleshoot estimation problems. For more information, see “Analyze Estimated Frequency Response” on page 5-38.

Note The **Linear Analysis Tool** only collects diagnostic information when **Diagnostic Viewer** is checked before performing an estimation task.

See Also

Functions

`findop` | `frestimate` | `linearize`

Topics

"Linearize Simulink Model at Model Operating Point" on page 2-69
"Linearize at Trimmed Operating Point" on page 2-85
"Linearize at Simulation Snapshot" on page 2-91
"Estimate Frequency Response Using Linear Analysis Tool" on page 5-26
"Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-29
"Analyze Results Using Linear Analysis Tool Response Plots" on page 2-146
"Batch Linearize Model for Parameter Value Variations Using Linear Analysis Tool" on page 3-75

Introduced in R2011b

addoutputspec

Add output specification to operating point specification

Syntax

```
newOpspec = addoutputspec(opspec,block,port)
```

Description

`newOpspec = addoutputspec(opspec,block,port)` adds an output specification for a Simulink model to an existing operating point specification or array of operating point specifications. The output specification is added for the signal that originates from the specified output `port` of a Simulink `block`.

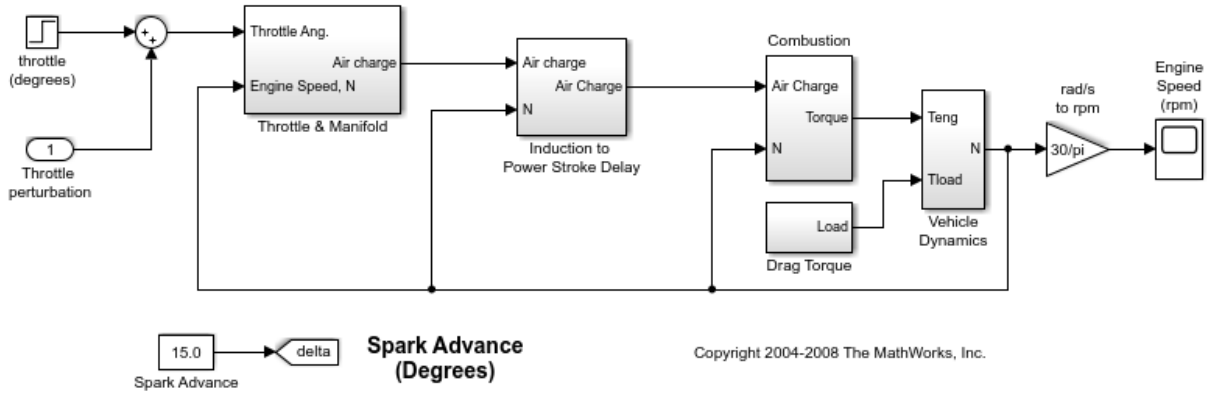
To find the width of the specified port, the `addoutputspec` command recompiles the model.

Examples

Add Output Specification to Operating Point Specification Object

Open the Simulink model.

```
sys = 'scdspeed';
open_system(sys)
```



Create a default operating point specification object for the model.

```
opspec = operspec(sys)


 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  initial guess: 209

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
     initial guess: 0

Outputs: None
----------
```

The default operating point specification object has no output specifications because there are no root-level outports in the model.

Add an output specification to the outport of the rad/s to rpm block.

```
newspec = addoutputspec(opspec,'scdspeed/rad//s to rpm',1);
```

Specify a known value of 2000 rpm for the output specification.

```
newspec.Outputs(1).Known = 1;
newspec.Outputs(1).y = 2000;
```

View the updated operating point specification.

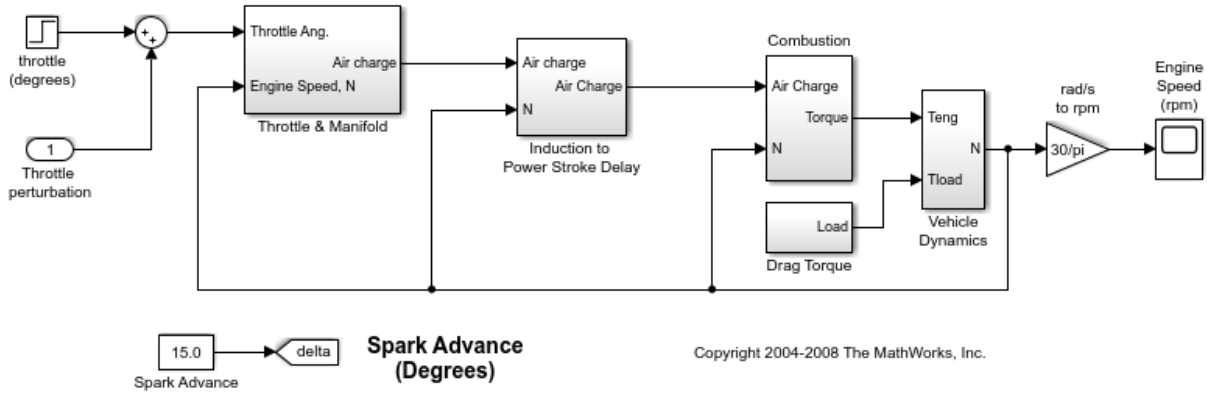
```
newspec
```

```
 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     spec:  dx = 0,  initial guess: 0.543
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     spec:  dx = 0,  initial guess: 209

Inputs:
----------
(1.) scdspeed/Throttle  perturbation
     initial guess: 0

Outputs:
----------
(1.) scdspeed/rad//s to rpm
     spec:  y = 2e+03
```

Add Output Specification to Multiple Operating Point Specification Objects

Open the Simulink model.

```
sys = 'scdspeed';
open_system(sys)
```



Create a 3-by-1 array of default operating point specification objects for the model.

```
opspec = operspec(sys,[3,1])
```

```
Array of operating point specifications for the model scdspeed. To display an
 operating point specification, select an element from the array.
```

Add an output specification to the outport of the rad/s to rpm block.

```
newspec = addoutputspec(opspec,'scdspeed/rad//s to rpm',1);
```

This output specification is added to all of the operating point specification objects in `opspec`.

You can specify different output constraints for each specification in `opspec`. For example, specify different known values for each specification.

```
newspec(1,1).Outputs(1).Known = 1;
newspec(1,1).Outputs(1).y = 1900;

newspec(2,1).Outputs(1).Known = 1;
newspec(2,1).Outputs(1).y = 2000;
```

```
newspec(3,1).Outputs(1).Known = 1;
newspec(3,1).Outputs(1).y = 2100;
```

Input Arguments

`opspec` — Operating point specification

`operspec` object | array of `operspec` objects

Operating point specification for a Simulink model, specified as one of the following:

- `operspec` object — Add output specification to a single `operspec` object.
- Array of `operspec` objects — Add the same output specification to all `operspec` objects in the array. All the specification objects must have the same `Model` property.

To create an operating point specification object for your model, use the `operspec` command.

`block` — Simulink block

character vector | string

Simulink block to which to add the output specification, specified as a character vector or string that contains its block path. The `block` must be in the Simulink model specified in `opspec.Model`.

`port` — Output port

positive integer

Output port to which to add the output specification, specified as a positive integer in the range [`1`,`N`], where `N` is the number of output ports on the specified `block`.

Output Arguments

`newOpspec` — Updated operating point specification

`operspec` object | array of `operspec` objects

Updated operating point specification, returned as an `operspec` object or an array of `operspec` objects with the same dimensions as `opspec`. `newOpspec` is the same as `opspec`, except that it contains the new output specification in its `Outputs` array.

You can modify the constraints and specifications for the new output specification using dot notation.

Alternative Functionality

Linear Analysis Tool

You can interactively add output specifications when trimming your model using the Linear Analysis Tool. For more information, see “Compute Steady-State Operating Point from Output Specifications” on page 1-28.

Simulink Model

You can add output specifications directly in your Simulink model. To do so, right-click the signal to which you want to add the specification, and select **Linear Analysis Points > Trim Output Constraint**.

See Also

`findop` | `operpoint` | `operspec`

Introduced before R2006a

advise

Package: linearize.advisor

Find blocks that are potentially problematic for linearization

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To search the `LinearizationAdvisor` object for diagnostics of blocks that are potentially problematic for linearization, use the `advise` function.

Syntax

```
advise(advisor)
result = advise(advisor)
```

Description

`advise(advisor)` opens the Linear Analysis Tool with an **Advisor** tab open for troubleshooting the block linearizations in `advisor`. For more information, see “Troubleshoot Linearization Results in Linear Analysis Tool” on page 4-21.

`result = advise(advisor)` returns a `LinearizationAdvisor` object that contains linearization diagnostic information for any blocks in `advisor` that are potentially problematic for linearization.

Examples

Open Linearization Advisor

Load Simulink model.

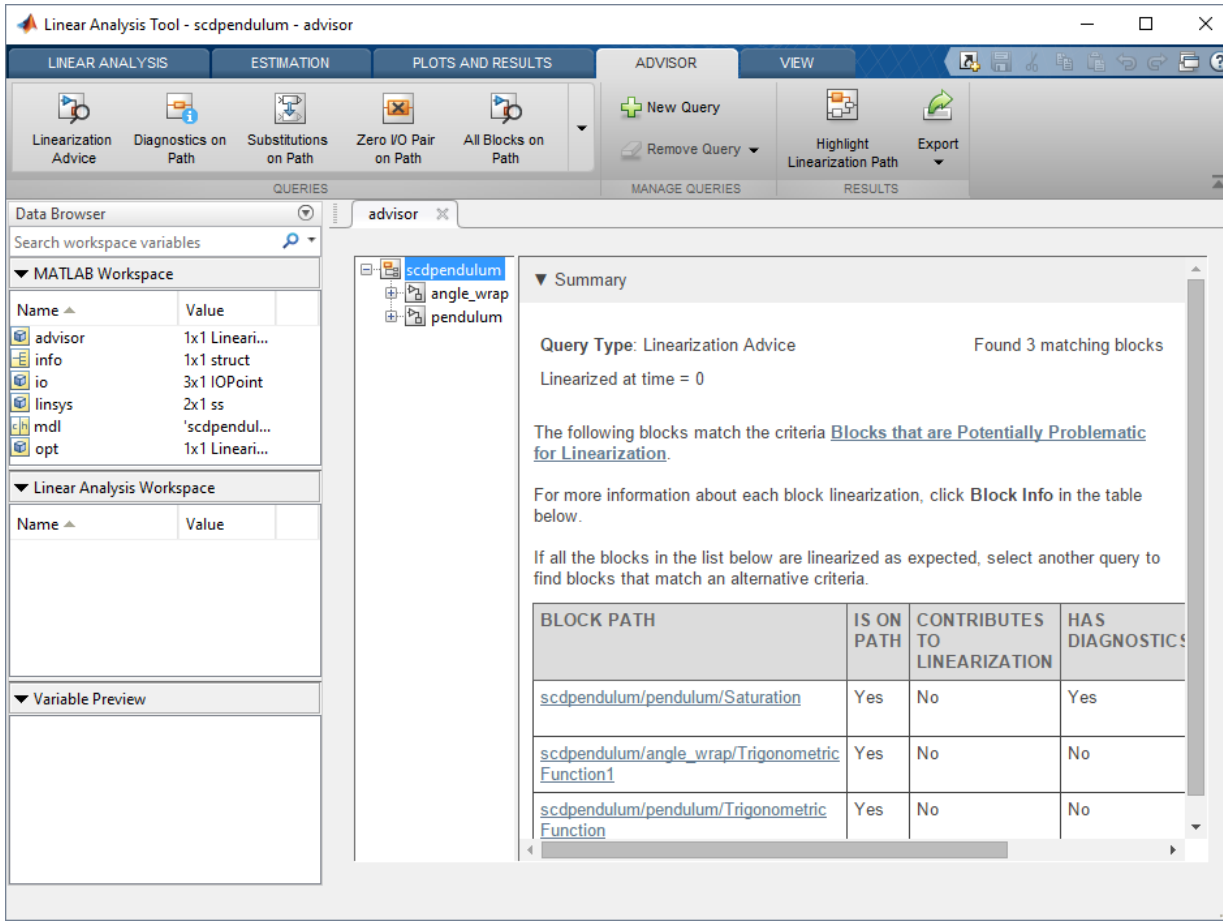
```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Open the Linearization Advisor in the Linear Analysis Tool.

```
advise(advisor)
```



Find Potentially Problematic Blocks for Linearization

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find potentially problematic blocks for linearization.

```
result = advise(advisor)
```

```
result =
  LinearizationAdvisor with properties:

              Model: 'scdpendulum'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
          QueryType: 'Linearization Advice'
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

Output Arguments

`result` — Diagnostic information for potentially problematic blocks

`LinearizationAdvisor` object

Diagnostic information for potentially problematic blocks in linearization results, returned as a `LinearizationAdvisor` object. `result` contains linearization diagnostic information for any blocks in `advisor` that are on the linearization path and satisfy at least one of the following criteria:

- Have diagnostic messages regarding the block linearization
- Linearize to zero
- Have substituted linearizations

Algorithms

Calling the `advise` function is equivalent to performing the following custom query with the `find` function:

```
qPath     = linqueryIsOnPath;
qZero     = linqueryIsZero;
qBlkRep   = linqueryIsBlockSubstituted;
qDiags    = linqueryHasDiagnostics;

q = qPath & (qZero | qDiags | qBlkRep);

advisor_new = find(advisor,q);
```

See Also

Apps
Linear Analysis Tool

Functions
`find` | `getBlockInfo` | `getBlockPaths` | `highlight`

Using Objects
`LinearizationAdvisor`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-40
“Identify and Fix Common Linearization Issues” on page 4-8

Introduced in R2017b

copy

Copy operating point or operating point specification

Syntax

```
op_point2=copy(op_point1)
op_spec2=copy(op_spec1)
```

Description

`op_point2=copy(op_point1)` returns a copy of the operating point object `op_point1`. You can create `op_point1` with the function `operpoint`.

`op_spec2=copy(op_spec1)` returns a copy of the operating point specification object `op_spec1`. You can create `op_spec1` with the function `operspec`.

Note The command `op_point2=op_point1` does not create a copy of `op_point1` but instead creates a pointer to `op_point1`. In this case, any changes made to `op_point2` are also made to `op_point1`.

Examples

Create an operating point object for the model, `magball`.

```
opp=operpoint('magball')
```

The operating point is displayed.

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
```

```
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

Create a copy of this object, `opp`.

```
new_opp=copy(opp)
```

An exact copy of the object is displayed.

```
Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

See Also

`operpoint` | `operspec`

Introduced before R2006a

find

Package: linearize.advisor

Find blocks in linearization results that match specific criteria

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Syntax

```
result = find(advisor,query)
```

Description

`result = find(advisor,query)` returns the subset of block diagnostics in `advisor` that match the search criteria specified in `query`.

Examples

Find Blocks on Linearization Path

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
```

```
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,~,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create a query object for finding blocks on the linearization path.

```
query = linqueryIsOnPath;
```

Find blocks using query object.

```
advOnPath = find(advisor,query)
```

```
advOnPath =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x26 linearize.advisor.BlockDiagnostic]
           QueryType: 'On Linearization Path'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

`query` — Search criteria

`CompoundQuery` object | `linqueryIsOnPath` object | `linqueryHasDiagnostics` object | `linqueryHasOrder` object | ...

Search criteria, specified as one of the following query objects or a logical combination of query objects (`CompoundQuery` object).

Query Object	Find Blocks That...
`linqueryAdvise`	Are potentially problematic for linearization.
`linqueryAllBlocks`	Are in the `advisor` object.
`linqueryContributesToLinearization`	Numerically contribute to the model linearization result.
`linqueryHasDiagnostics`	Have diagnostic messages regarding their linearization.
`linqueryHasInputs`	Have a specified number of inputs.
`linqueryHasOrder`	Have a specified number of states.
`linqueryHasOutputs`	Have a specified number of outputs.
`linqueryHasSampleTime`	Have a specified sample time.

Query Object	Find Blocks That...
`linqueryHasZeroIOPair`	Have at least one input/output pair that linearizes to zero.
`linqueryIsBlockSubstituted`	Have a custom block linearization specified.
`linqueryIsBlockType`	Are of a specified type.
`linqueryIsExact`	Are linearized using their defined exact linearization.
`linqueryIsNumericallyPerturbed`	Are linearized using numerical perturbation.
`linqueryIsOnPath`	Are on the linearization path.
`linqueryIsZero`	Linearize to zero.

To create a compound query, combine these queries using AND (`&`), OR (`|`), and NOT (`~`) logical operations. For example, to find all blocks on the linearization path that do not contribute to the model linearization result, use:

```
compundQuery = linqueryIsOnPath & ~linqueryContributesToLinearization
```

Output Arguments

`result` — Diagnostic information for blocks that match the search criteria

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for blocks that match the search criteria specified in `query`, returned as:

- `LinearizationAdvisor` object if `advisor` is a single `LinearizationAdvisor` object.
- A `LinearizationAdvisor` object with the same dimensions as `advisor` if `advisor` is an array.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`advise` | `getBlockInfo` | `getBlockPaths` | `highlight`

Topics

"Troubleshoot Linearization Results at Command Line" on page 4-40

Introduced in R2017b

findop

Steady-state operating point from specifications (trimming) or simulation

Syntax

```
op = findop(sys,opspec)
op = findop(sys,opspec,param)

op = findop(___,options)

[op,opreport] = findop(___)

op = findop(sys,tsnapshot)
op = findop(sys,tsnapshot,param)
```

Description

`op = findop(sys,opspec)` returns the operating point of the model that meets the specifications in `opspec`. Typically, you trim the model at a steady-state operating point on page 13-41. The Simulink model must be open. If `opspec` is an array of operating points specifications, `findop` returns an array of corresponding operating points.

`op = findop(sys,opspec,param)` batch trims the model for the parameter value variations specified in `param`.

`op = findop(___,options)` trims the model using additional optimization algorithm `options`.

`[op,opreport] = findop(___)` returns an operating point search report, `opreport`, for any of the previous syntaxes.

`op = findop(sys,tsnapshot)` simulates the model using the model initial conditions, and extracts operating points at simulation snapshot times specified in `tsnapshot`.

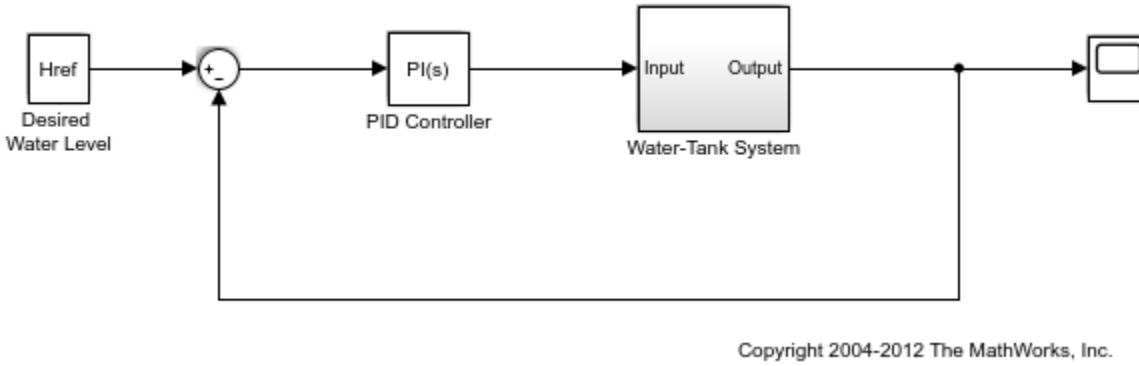
`op = findop(sys,tsnapshot,param)` simulates the model and extracts operating points at simulation snapshot times.

Examples

Trim Model to Meet State Specifications

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Trim the model to find a steady-state operating point where the water tank level is `10`.

Create default operating point specification object.

```
opspec = operspec(sys);
```

Configure specifications for the first model state. The first state must be at steady state with a lower bound of `0`. Provide an initial guess of `2` for the state value.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Configure the second model state as a known state with a value of `10`.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

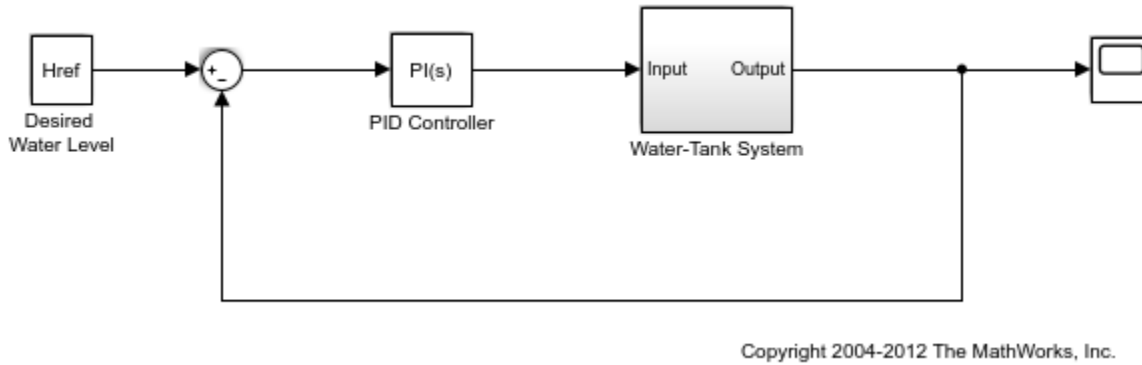
Find the operating point that meets these specifications.

```
op = findop(sys,opspec);


 Operating point search report:
---------------------------------

 Operating point search report for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) watertank/PID Controller/Integrator
      x:           1.26      dx:             0 (0)
(2.) watertank/Water-Tank System/H
      x:             10      dx:             0 (0)

Inputs: None
----------

Outputs: None
----------
```

Batch Trim Simulink Model for Parameter Variation

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Vary parametsrs `A` and `b` within 10% of their nominal values, and create a 3-by-4 parameter grid.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the model.

```
opspec = operspec(sys);
```

Trim the model using the specified operating point specification and parameter grid.

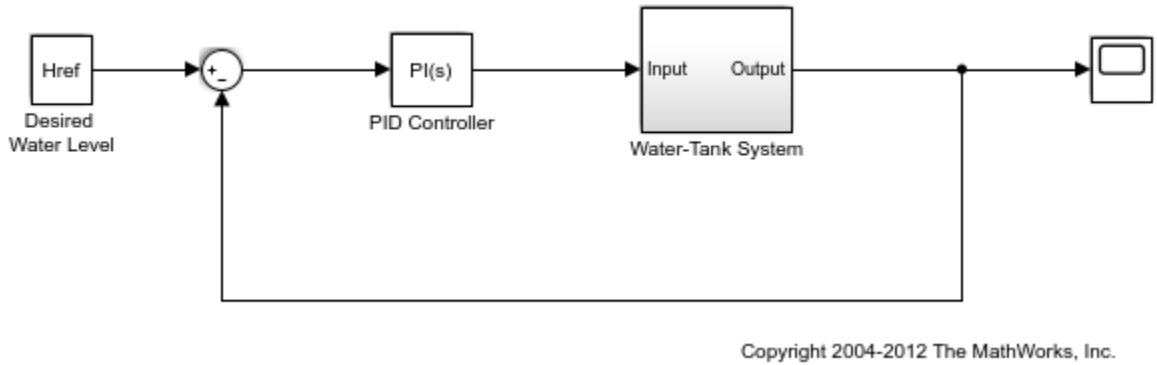
```
opt = findopOptions('DisplayReport','off');
op = findop(sys,opspec,params,opt);
```

`op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

Trim Model Using Specified Optimizer Type

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create a default operating point specification object.

```
opspec = operspec(sys);
```

Create an option set that sets the optimizer type to gradient descent and suppresses the search report display.

```
opt = findopOptions('OptimizerType','graddescent','DisplayReport','off');
```

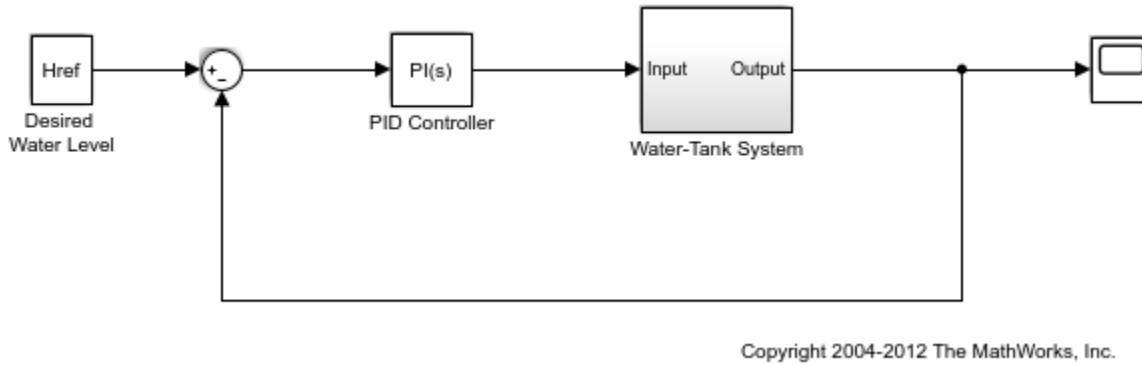
Trim the model using the specified option set.

```
op = findop(sys,opspec,opt);
```

Obtain Operating Point Search Report

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create default operating point specification object.

```
opspec = operspec(sys);
```

Configure specifications for the first model state.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Configure specifications for the second model state.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

Find the operating point that meets these specifications, and return the operating point search report. Create an option set to suppress the search report display.

```
opt = findopOptions('DisplayReport',false);
[op,opreport] = findop(sys,opspec,opt);
```

`opreport` describes how closely the optimization algorithm met the specifications at the end of the operating point search.

```
opreport
```

```
 Operating point search report for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)
```

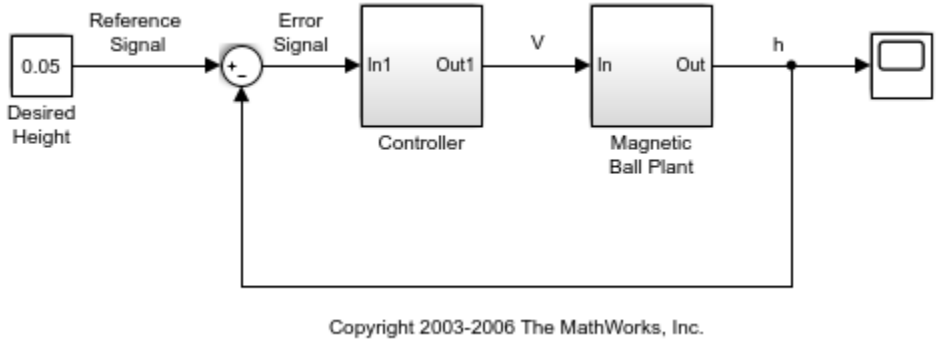
```
Operating point specifications were successfully met.
States:
----------
(1.) watertank/PID Controller/Integrator
      x:          1.26      dx:             0 (0)
(2.) watertank/Water-Tank System/H
      x:            10      dx:             0 (0)

Inputs: None
----------

Outputs: None
----------
```

`dx` is the time derivative for each state. Since all `dx` values are zero, the operating point is at steady state.

Extract Operating Points at Simulation Snapshots

Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```



Simulate the model, and extract operating points at `10` and `20` time units.

```
op = findop(sys,[10,20]);
```

`op` is a column vector of operating points, with one element for each snapshot time.

Display the first operating point.

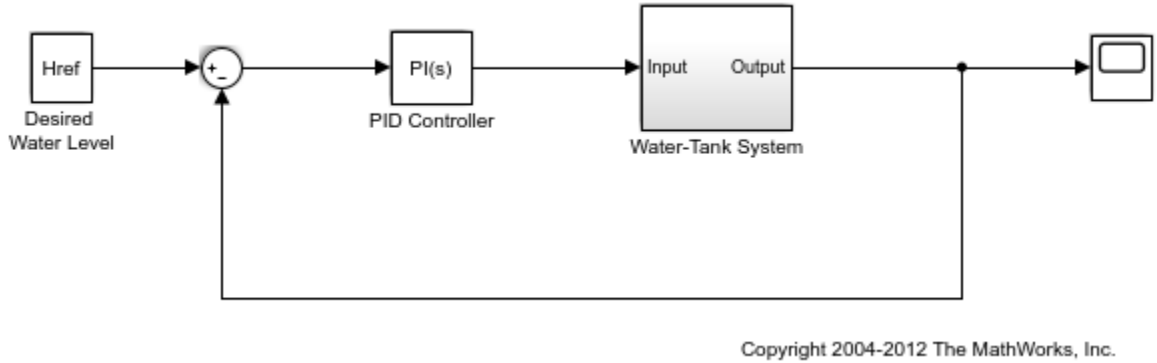
```
op(1)
```

```
 Operating point for the Model magball.
 (Time-Varying Components Evaluated at time t=10)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 5.47e-07
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 8.44e-08
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

Vary Parameters and Extract Operating Points at Simulaton Snapshots

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Specify parameter values. The parameter grids are 5-by-4 arrays.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,5),...
                         linspace(0.9*b,1.1*b,4));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Simulate the model and extract operating points at `0`, `5`, and `10` time units.

```
op = findop(sys,[0 5 10],params);
```

`findop` simulates the model for each parameter value combination, and extracts operating points at the specified simulation times.

`op` is a 3-by-5-by-4 array of operating point objects.

```
size(op)


ans =
```

```
    3     5     4
```

Input Arguments

`sys` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

`opspec` — Operating point specifications

`operspec` object | array of `operspec` objects

Operating point specifications for trimming the model, specified as an `operspec` object or an array of `operspec` objects.

If `opspec` is an array, `findop` returns an array of corresponding operating points using a single model compilation.

`param` — Parameter samples

structure | structure array

Parameter samples for trimming, specified as one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,3));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

When you specify parameter value variations, `findop` batch trims the model for each parameter value combination, and returns an array of corresponding operating points. If `param` specifies tunable parameters only, then the software batch trims the model using a single compilation.

If you specify `opspec` as a single `operspec` object and the parameter values in `param` produce states that conflict with known states in `opspec`, `findop` trims the model using the specifications in `opspec`. To trim the model at state values derived from the parameter values, specify `opspec` as an array of corresponding `operspec` objects. For an example, see “Batch Trim Simulink Model for Parameter Variation” on page 13-28.

`options` — Trimming options

`findopOptions` option set

Trimming options, specified as a `findopOptions` option set.

`tsnapshot` — Simulation snapshot times

scalar | vector

Simulation snapshot times at which to extract the operating point of the model, specified as a scalar for a single snapshot or a vector for multiple snapshots. `findop` simulates the model and computes an operating point for the state of the model at each snapshot time.

Output Arguments

`op` — Operating point

operating point object | array of operating point objects

Operating point, returned as an operating point object or an array of operating point objects. The dimensions of `op` depend on the specified parameter variations and either the operating point specifications or the simulation snapshot time.

Parameter Variation	Find operating point for...	Resulting `op` Dimensions
No parameter variation	Single operating point specification, specified by `opspec`	single operating point object
	Single snapshot time, specified by `tsnapshot`	
	N_1-by-. . .-by-N_m array of operating point specifications, specified by `opspec`	N_1-by-. . .-by-N_m
	N_s snapshots, specified by `tsnapshot`	Column vector of length N_s
N_1-by-. . .-by-N_m parameter grid, specified by `param`	Single operating point specification, specified by `opspec`	N_1-by-. . .-by-N_m
	Single snapshot time, specified by `tsnapshot`	
	N_1-by-. . .-by-N_m array of operating point specifications, specified by `opspec`	
	N_s snapshots, specified by `tsnapshot`	N_s-by-N_1-by-. . .-by-N_m.

For example, suppose:

- `opspec` is a single operating point specification object and `param` specifies a 3-by-4-by-2 parameter grid. In this case, `op` is a 3-by-4-by-2 array of operating points.
- `tsnapshot` is a scalar and `param` specifies a 5-by-6 parameter grid. In this case, `op` is a 1-by-5-by-6 array of operating points.
- `tsnapshot` is a row vector with three elements and `param` specifies a 5-by-6 parameter grid. In this case, `op` is a 3-by-5-by-6 array of operating points.

Each operating point object has the following properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Model</td><td>Simulink model name, returned as a character vector.</td></tr>
<tr><td>States</td><td>State operating point, returned as a vector of state objects. Each entry in States represents the supported states of one Simulink block.

For a list of supported states for operating point objects, see “Simulink Model States Included in Operating Point Object” on page 1-4.

Note If the block has multiple named continuous states, States contains one structure for each named state.

Each state object has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nx (read only)</td><td>Number of states in the block</td></tr>
<tr><td>Block</td><td>Block path, returned as a character vector.</td></tr>
<tr><td>StateName</td><td>State name</td></tr>
<tr><td>x</td><td>Values of all supported block states, returned as a vector of length Nx.</td></tr>
<tr><td>Ts</td><td>Sample time and offset of each supported block state, returned as a vector. For continuous-time systems, Ts is zero.</td></tr>
<tr><td>SampleType</td><td>State time rate, returned as one of the following:
• 'CSTATE' — Continuous-time state
• 'DSTATE' — Discrete—time state</td></tr>
<tr><td>inReferencedModel</td><td>Flag indicating whether the block is inside a reference model, returned as one of the following:
• 1 — Block is inside a reference model.
• 0 — Block is in the current model file.</td></tr>
<tr><td>Description</td><td>Block state description, returned as a character vector.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Inputs</td><td>Input level at the operating point, returned as a vector of input objects. Each entry in Inputs represents the input levels of one root-level inport block in the model.

Each entry input object has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Block</td><td>Inport block name</td></tr>
<tr><td>PortWidth</td><td>Number of inport block signals</td></tr>
<tr><td>PortDimensions</td><td>Dimension of signals accepted by the inport</td></tr>
<tr><td>u</td><td>Inport block input levels at the operating point, returned as a vector of length PortWidth.</td></tr>
<tr><td>Description</td><td>Inport block input description, returned as a character vector.</td></tr>
</table></td></tr>
<tr><td>Time</td><td>Times at which any time-varying functions in the model are evaluated, returned as a vector.</td></tr>
<tr><td>Version</td><td>Object version number</td></tr>
</table>

You can edit the properties of op using dot notation or the set function.

opreport — Operating point search report

operating point search report object | array of operating point search report objects

Operating point search report, returned as an operating point search report object. If op is an array of operating point objects, then opreport is an array of corresponding search reports.

This report displays automatically, even when you suppress the output using a semicolon. To hide the report, set the DisplayReport field in options to 'off'.

Each operating point search report has the following properties:

Property	Description
Model	Model property value of op
Inputs	Inputs property value of op

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Outputs</td><td>Outputs property value of op, with the addition of yspec, which is the desired y value</td></tr>
<tr><td>States</td><td>States property value of op with the addition of dx, which contains the state derivative values. For discrete-time states, dx is the difference between the next state value and the current one; that is, $x(k+1) - x(k)$.</td></tr>
<tr><td>Time</td><td>Time property value of op</td></tr>
<tr><td>TerminationString</td><td>Optimization termination condition, returned as a character vector.</td></tr>
<tr><td>OptimizationOutput</td><td>Optimization algorithm search results, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>iterations</td><td>Number of iterations performed during the optimization</td></tr>
<tr><td>funcCount</td><td>Number of function evaluations performed during the optimization</td></tr>
<tr><td>lssteplength</td><td>Size of line search step relative to search direction (active-set optimization algorithm only)</td></tr>
<tr><td>stepsize</td><td>Displacement in the state vector at the final iteration (active-set and interior-point optimization algorithms)</td></tr>
<tr><td>algorithm</td><td>Optimization algorithm used</td></tr>
<tr><td>firstorderopt</td><td>Measure of first-order optimization, for the trust-region-reflective optimization algorithm; [] for other algorithms</td></tr>
<tr><td>constrviolation</td><td>Maximum of constraint functions</td></tr>
<tr><td>message</td><td>Exit message</td></tr>
</table>
For more information about the optimization algorithm, see the Optimization Toolbox documentation.</td></tr>
</table>

Definitions

Steady-State Operating Point (Trim Condition)

A *steady-state operating point* of a model, also called an equilibrium or *trim* condition, includes state variables that do not change with time.

A model can have several steady-state operating points. For example, a hanging damped pendulum has two steady-state operating points at which the pendulum position does not change with time. A *stable steady-state operating point* occurs when a pendulum hangs straight down. When the pendulum position deviates slightly, the pendulum always returns to equilibrium. In other words, small changes in the operating point do not cause the system to leave the region of good approximation around the equilibrium value.

An *unstable steady-state operating point* occurs when a pendulum points upward. As long as the pendulum points *exactly* upward, it remains in equilibrium. However, when the pendulum deviates slightly from this position, it swings downward and the operating point leaves the region around the equilibrium value.

When using optimization search to compute operating points for nonlinear systems, your initial guesses for the states and input levels must be near the desired operating point to ensure convergence.

When linearizing a model with multiple steady-state operating points, it is important to have the right operating point. For example, linearizing a pendulum model around the stable steady-state operating point produces a stable linear model, whereas linearizing around the unstable steady-state operating point produces an unstable linear model.

Tips

- You can initialize an operating point search at a simulation snapshot or a previously computed operating point using `initopspec`.
- Linearize the model at the operating point `op` using `linearize`.

Algorithms

By default, `findop` uses the optimizer `graddescent-elim`. To use a different optimizer, change the value of `OptimizerType` in `options` using `findopOptions`.

`findop` automatically sets these Simulink model properties for optimization:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`
- `SaveFormat = 'StructureWithTime'`

After the optimization completes, Simulink restores the original model properties.

Alternatives

As an alternative to the `findop` command, you can find operating points using the Linear Analysis Tool. For more information, see the following examples:

- "Compute Steady-State Operating Point from State Specifications" on page 1-14
- "Compute Steady-State Operating Point from Output Specifications" on page 1-28

See Also

`addoutputspec` | `findopOptions` | `initopspec` | `linearize` | `operspec`

Topics

"About Operating Points" on page 1-2
"Compute Steady-State Operating Points" on page 1-6
"Compute Operating Points at Simulation Snapshots" on page 1-78

Introduced before R2006a

findopOptions

Set options for finding operating points from specifications

Syntax

```
options = findopOptions
options = findopOptions(Name,Value)
```

Description

`options = findopOptions` returns the default operating point search options.

`options = findopOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments. Use this option set to specify options for the `findop` command.

Examples

Create Option Set for Operating Point Search

Create an option set for operating point search that sets the optimizer type to gradient descent and suppresses the display output of `findop`.

```
option = findopOptions('OptimizerType','graddescent','DisplayReport','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = findopOptions;
options.OptimizerType = 'graddescent';
options.DisplayReport = 'off';
```

Input Arguments

Name-Value Pair Arguments

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

Example: `'DisplayReport','off'` suppresses the display of the operating point search report to the Command Window.

`OptimizerType` — Optimizer type used by the optimization algorithm

`'graddescent-elim'` (default) | `'graddescent'` | `'graddescent-proj'` | `'lsqnonlin'` | `'lsqnonlin-proj'` | `'simplex'`

Optimizer type used by the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerType'` and one of the following:

- `'graddescent-elim'` — Enforce an equality constraint to force the time derivatives of states to be zero (`dx/dt = 0`, `x(k+1) = x(k)`) and output signals to be equal to their specified known values. The optimizer fixes the states, `x`, and inputs, `u`, that are marked as `Known` in an operating point specification, and optimizes the remaining variables.
- `'graddescent'` — Enforce an equality constraint to force the time derivatives of states to be zero (`dx/dt = 0`, `x(k+1) = x(k)`) and the output signals to be equal to their specified known values. The optimizer also minimizes the error between the states, `x`, and inputs, `u`, and their respective known values from an operating point specification. If there are not any inputs or states marked as `Known`, `findop` attempts to minimize the deviation between the initial guesses for `x` and `u`, and their trimmed values.
- `'graddescent-proj'` — In addition to `'graddescent'`, enforce consistency of model initial conditions at each function evaluation. To specify whether constraints

are hard or soft, use the `ConstraintType` option. This optimization method does not support analytical Jacobians.

- `'lsqnonlin'` — Fix the states, `x`, and inputs, `u`, marked as `Known` in an operating point specification, and optimize the remaining variables. The algorithm tries to minimize both the error in the time derivatives of the states (`dx/dt = 0`, `x(k+1) = x(k)`) and the error between the outputs and their specified known values.
- `'lsqnonlin-proj'` — In addition to `'lsqnonlin'`, enforce consistency of model initial conditions at each function evaluation. This optimization method does not support analytical Jacobians.
- `'simplex'` — Use the same cost function as `lsqnonlin` with the direct search optimization routine found in `fminsearch`.

For more information about these optimization algorithms, see `fmincon`, `lsqnonlin`, and `fminsearch` in the Optimization Toolbox documentation.

`OptimizationOptions` — Options for the optimization algorithm
structure

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizationOptions'` and a structure created using the `optimset` function.

`DisplayReport` — Flag indicating whether to display the operating summary report
`'on'` (default) | `'off'` | `'iter'`

Flag indicating whether to display the operating point summary report, specified as the comma-separated pair consisting of `'DisplayReport'` and one of the following:

- `'on'` — Display the operating point summary report in the MATLAB command window when running `findop`.
- `'off'` — Suppress display of the summary report.
- `'iter'` — Display an iterative update of the optimization progress.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values
`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for trimming, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.
- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

`ConstraintType` — Constraint types for `'graddescent-proj'`
structure

Constraint types for `'graddescent-proj'` optimizer algorithm, specified as the comma-separated pair consisting of `'ConstraintType'` and a structure with the following fields:

- `dx` — Type for constraints on state derivatives
- `x` — Type for constraints on state values
- `y` — Type for constraints on output values

Specify each constraint as one of the following:

- `'hard'` — Enforce the constraints to be zero.
- `'soft'` — Minimize the constraints.

All constraint types are `'hard'` by default.

Output Arguments

`options` — Trimming options
`findopOptions` option set

Trimming options, returned as a `findopOptions` option set.

See Also

`findop`

Introduced in R2013b

frest.Chirp

Package: frest

Swept-frequency cosine signal

Syntax

```
input = frest.Chirp(sys)
input = frest.Chirp('OptionName',OptionValue)
```

Description

`input = frest.Chirp(sys)` creates a swept-frequency cosine input signal based on the dynamics of a linear system *sys*.

`input = frest.Chirp('OptionName',OptionValue)` creates a swept-frequency cosine input signal using the options specified by comma-separated name/value pairs.

To view a plot of your input signal, type `plot(input)`. To obtain a timeseries for your input signal, use the `generateTimeseries` command.

Input Arguments

`sys`

Linear system for creating a chirp signal based on the dynamic characteristics of this system. You can specify the linear system based on known dynamics using `tf`, `zpk`, or `ss`. You can also obtain the linear system by linearizing a nonlinear system.

The resulting chirp signal automatically sets these options based on the linear system:

- `'FreqRange'` are the frequencies at which the linear system has interesting dynamics.

- `'Ts'` is set to avoid aliasing such that the Nyquist frequency of the signal is five times the upper end of the frequency range.
- `'NumSamples'` is set such that the frequency response estimation includes the lower end of the frequency range.

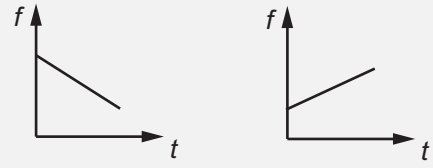
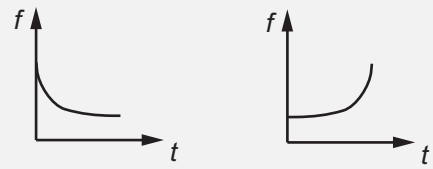
Other chirp options have default values.

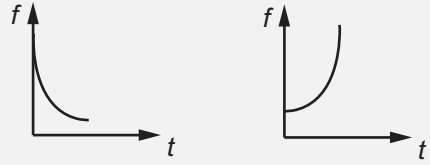
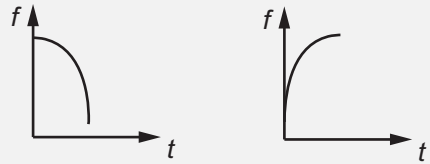
`'OptionName',OptionValue`

Signal characteristics, specified as comma-separated option name and option value pairs.

Option Name	Option Value
`'Amplitude'`	Signal amplitude. **Default:** `1e-5`
`'FreqRange'`	Signal frequencies, specified as either: • Two-element vector, for example `[w1 w2]` • Two-element cell array, for example `{w1 w2}` **Default:** `[1,1000]`
`'FreqUnits'`	Frequency units: • `'rad/s'` — Radians per second • `'Hz'` — Hertz Changing frequency units does not impact frequency response estimation. **Default:** `'rad/s'`
`'Ts'`	Sample time of the chirp signal in seconds. The default setting avoids aliasing. **Default:** $\frac{2\pi}{5 * \max(FreqRange)}$

Option Name	Option Value
`'NumSamples'`	Number of samples in the chirp signal. Default setting ensures that the estimation includes the lower end of the frequency range. **Default:** $\frac{4\pi}{Ts * \min(FreqRange)}$

Option Name	Option Value
`'SweepMethod'`	Method for evolution of instantaneous frequency: • `'linear'` (default) — Specifies the instantaneous frequency sweep $f_i(t)$: $f_i(t) = f_0 + \beta t \quad where \quad \beta = (f_1 - f_0) / t_f$ ß ensures that the signal maintains the desired frequency breakpoint f_1 at final time t_f. **f1 > f2** **f1 < f2**  • `'logarithmic'` — Specifies the instantaneous frequency sweep $f_i(t)$ given by $f_i(t) = f_0 \times \beta^t \quad where \quad \beta = \left(\frac{f_1}{f_0}\right)^{\frac{1}{t_f}}$ **f1 > f2** **f1 < f2**  • `'quadratic'` — Specifies the instantaneous frequency sweep $f_i(t)$: $f_i(t) = f_0 + \beta t^2 \quad where \quad \beta = (f_1 - f_0) / t_i^2$ Also specify the shape of the quadratic using the `'Shape'` option.

Option Name	Option Value
`'Shape'`	Use when you set `'SweepMethod'` to `'quadratic'` to describe the shape of the parabola in the positive frequency axis: • `'concave'` — Concave quadratic sweeping shape. **f1 > f2** **f1 < f2**  • `'convex'` — Convex quadratic sweeping shape. **f1 > f2** **f1 < f2** 
`'InitialPhase'`	Initial phase of the Chirp signal in degrees. **Default:** `270`

Examples

Create a chirp input signal:

```
input = frest.Chirp('Amplitude',1e-3,'FreqRange',[10 500],'NumSamples',20000)
```

See Also

`frest.Random` | `frest.Sinestream` | `frestimate` | `generateTimeseries` | `getSimulationTime`

Topics

“Estimation Input Signals” on page 5-7
“Estimate Frequency Response at the Command Line” on page 5-33
“Estimate Frequency Response Using Linear Analysis Tool” on page 5-26

Introduced in R2009b

frest.createFixedTsSinestream

Package: frest

Sinestream input signal with fixed sample time

Syntax

```
input = frest.createFixedTsSinestream(ts)
input = frest.createFixedTsSinestream(ts,{wmin wmax})
input = frest.createFixedTsSinestream(ts,w)
input = frest.createFixedTsSinestream(ts,sys)
input = frest.createFixedTsSinestream(ts,sys,{wmin wmax})
input = frest.createFixedTsSinestream(ts,sys,w)
```

Description

`input = frest.createFixedTsSinestream(ts)` creates sinestream input signal in which each frequency has the same fixed sample time `ts` in seconds. The signal has 30

frequencies between 1 and ω_s, where $\omega_s = \frac{2\pi}{t_s}$ is the sample rate in radians per second. The software adjusts the `SamplesPerPeriod` option to ensure that each frequency has the same sample time. Use when your Simulink model has linearization input I/Os on signals with discrete sample times.

`input = frest.createFixedTsSinestream(ts,{wmin wmax})` creates sinestream input signal with up to 30 frequencies logarithmically spaced between `wmin` and `wmax` in radians per second.

`input = frest.createFixedTsSinestream(ts,w)` creates sinestream input signal with frequencies `w`, specified as a vector of frequency values in radians per second. The

values of `w` must satisfy $w = \frac{2\pi}{Nts}$ for integer N such that the sample rate $\omega_s = \frac{2\pi}{t_s}$ is an integer multiple of each element of `w`.

`input = frest.createFixedTsSinestream(ts,sys)` creates sinestream input signal with a fixed sample time `ts`. The signal's frequencies, settling periods, and number of periods automatically set based on the dynamics of a linear system `sys`.

`input = frest.createFixedTsSinestream(ts,sys,{wmin wmax})` creates sinestream input signal with up to 30 frequencies logarithmically spaced between `wmin` and `wmax` in radians per second.

`input = frest.createFixedTsSinestream(ts,sys,w)` creates sinestream input signal at frequencies `w`, specified as a vector of frequency values in radians per second.

The values of `w` must satisfy $w = \frac{2\pi}{Nts}$ for integer N such that the sample rate `ts` is an integer multiple of each element of `w`.

Examples

Create a sinusoidal input signal with the following characteristics:

- Sample time of 0.02 sec
- Frequencies of the sinusoidal signal are between 1 rad/s and 10 rad/s

```
input = frest.createFixedTsSinestream(0.02,{1, 10});
```

See Also

`frest.Sinestream` | `frestimate`

Topics

“Estimation Input Signals” on page 5-7
“Estimate Frequency Response at the Command Line” on page 5-33
“Estimate Frequency Response Using Linear Analysis Tool” on page 5-26

Introduced in R2009b

frest.createStep

Package: frest

Step input signal

Syntax

```
input = frest.createStep('OptionName',OptionValue)
```

Description

`input = frest.createStep('OptionName',OptionValue)` creates a step input signal as a MATLAB `timeseries` object using the options specified by comma-separated name/value pairs.

Plot your input signal using `plot(input)`.

Input Arguments

`'OptionName',OptionValue`

Signal characteristics, specified as comma-separated option name and option value pairs.

Option Name	Option Value
`'Ts'`	Sample time of the step input in seconds. **Default**: `1e-3`
`'StepTime'`	Time in seconds when the output jumps from 0 to the `StepSize` parameter. **Default**: `1`
`'StepSize'`	Value of the step signal after time reaches and exceeds the `StepTime` parameter. **Default**: `1`

Option Name	Option Value
`'FinalTime`	Final time of the step input signal in seconds. **Default**: `10`

Examples

Create step signal:

```
input = frest.createStep('StepTime',3,'StepSize',2)
```

See Also

`frest.simCompare` | `frestimate`

Topics

"Estimation Input Signals" on page 5-7
"Estimate Frequency Response at the Command Line" on page 5-33
"Estimate Frequency Response Using Linear Analysis Tool" on page 5-26

Introduced in R2009b

frest.findDepend

Package: frest

List of model path dependencies

Syntax

`dirs = frest.findDepend(model)`

Description

`dirs = frest.findDepend(model)` returns paths containing Simulink model dependencies required for frequency response estimation using parallel computing. `model` is the Simulink model to estimate, specified as a character vector or a string. `dirs` is a cell array, where each element is a path character vector. `dirs` is empty when `frest.findDepend` does not detect any model dependencies. Append paths to `dirs` when the list of paths is empty or incomplete.

`frest.findDepend` does not return a complete list of model dependency paths when the dependencies are undetectable.

Examples

Specify model path dependencies for parallel computing:

```
% Copy referenced model to temporary folder.
pathToLib = scdpathdep_setup;

% Add folder to search path.
addpath(pathToLib);

% Open Simulink model.
mdl = 'scdpathdep';
open_system(mdl);

% Get model dependency paths.
dirs = frest.findDepend(mdl)
```

```
% The resulting path is on a local drive, C:/.
% Replace C:/ with valid network path accessible to remote workers.
dirs = regexprep(dirs,'C:/','\\\\hostname\\C$\\')

% Enable parallel computing and specify the model path dependencies.
options = frestimateOptions('UseParallel','on','ParallelPathDependencies',dirs)
```

See Also

`frestimate`

Topics

“Speeding Up Estimation Using Parallel Computing” on page 5-80
“Scope of Dependency Analysis” (Simulink)

Introduced in R2010a

frest.findSources

Package: frest

Identify time-varying source blocks

Syntax

```
blocks = frest.findSources(model)
blocks = frest.findSources(model,io)
```

Description

`blocks = frest.findSources(model)` finds all time-varying source blocks in the signal path of any linearization output point marked in the Simulink model `model`.

`blocks = frest.findSources(model,io)` finds all time-varying source blocks in the signal path of any linearization output point specified in the array of linear analysis points `io`.

Input Arguments

`model`

Character vector or string that contains the name of the Simulink model in which you are identifying time-varying source blocks for frequency response estimation.

`io`

Array of linearization I/O points.

The elements of `io` are linearization I/O objects that you create with `getlinio` or `linio`. `frest.findSources` uses only the output points to locate time-varying source blocks that can interfere with frequency response estimation. See “Algorithms” on page 13-64 for more information.

Output Arguments

`blocks`

Array of `Simulink.BlockPath` objects identifying the block paths of all time-varying source blocks in `model` that can interfere with frequency response estimation. The `blocks` argument includes time-varying source blocks inside subsystems and normal-mode referenced models.

If you provide `io`, `blocks` contains all time-varying source blocks contributing to the signal at the output points in `io`.

If you do not provide `io`, `blocks` contains all time-varying source blocks contributing to the signal at the output points marked in `model`.

Examples

Estimate the frequency response of a model having time-varying source blocks. This example shows the use of `frest.findSources` to identify time-varying source blocks that interfere with frequency response estimation. You can also see the use of `BlocksToHoldConstant` option of `frestimateOptions` to disable time-varying source blocks in the estimation.

Load the model `scdspeed_ctrlloop`.

```
mdl = 'scdspeed_ctrlloop';
open_system(mdl)
% Convert referenced model to normal mode for accuracy
set_param('scdspeed_ctrlloop/Engine Model',...
                    'SimulationMode','Normal');
```

First, view the effects of time-varying source blocks on frequency response estimation. To do so, perform the estimation without disabling time-varying source blocks.

In this example, linearization I/O points are already defined in the model. Use the `getlinio` command to get the I/O points for `frestimate`.

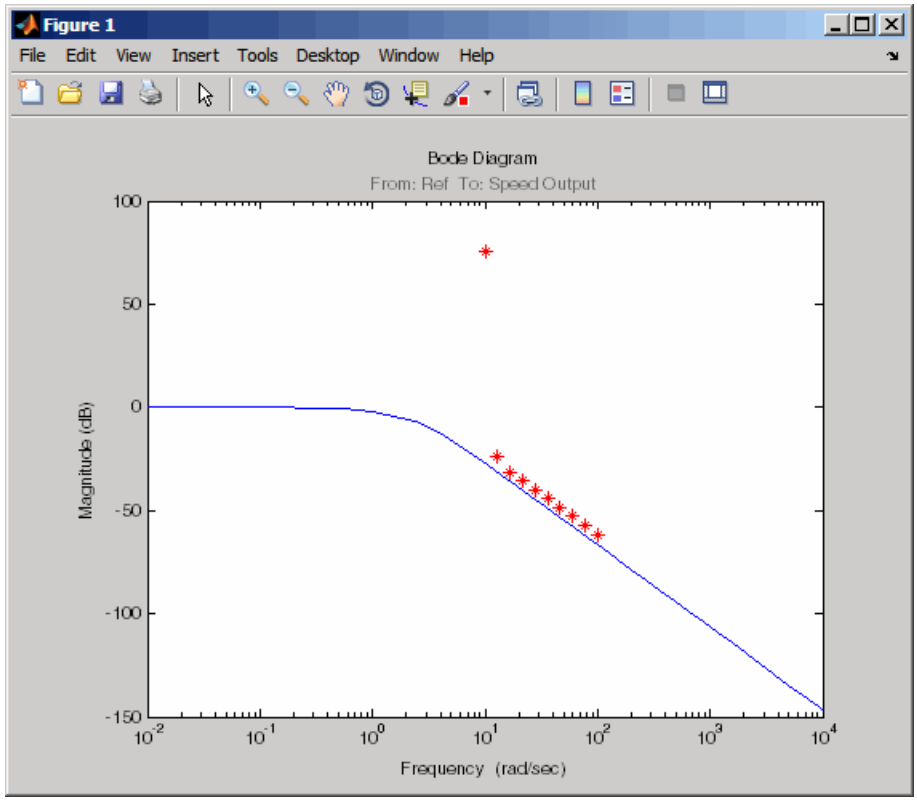
```
io = getlinio(mdl)
```

Define a sinestream signal and compute the estimated frequency response `sysest`.

```
in = frest.Sinestream('Frequency',logspace(1,2,10),...
              'NumPeriods',30,'SettlingPeriods',25);
[sysest,simout] = frestimate(mdl,io,in);
```

Perform exact linearization, and compare to the estimated response.

```
sys = linearize(mdl,io);
bodemag(sys,sysest,'r*')
```



The estimated frequency response does not match the exact linearization. The mismatch occurs because time-varying source blocks in the model prevent the response from reaching steady state.

Find the time-varying blocks using `frest.findSources`.

```
srcblks = frest.findSources(mdl);
```

`srcblks` is an array of block paths corresponding to the time-varying source blocks in the model. To examine the result, index into the array.

For example, entering

```
srcblks(2)
```

returns the result

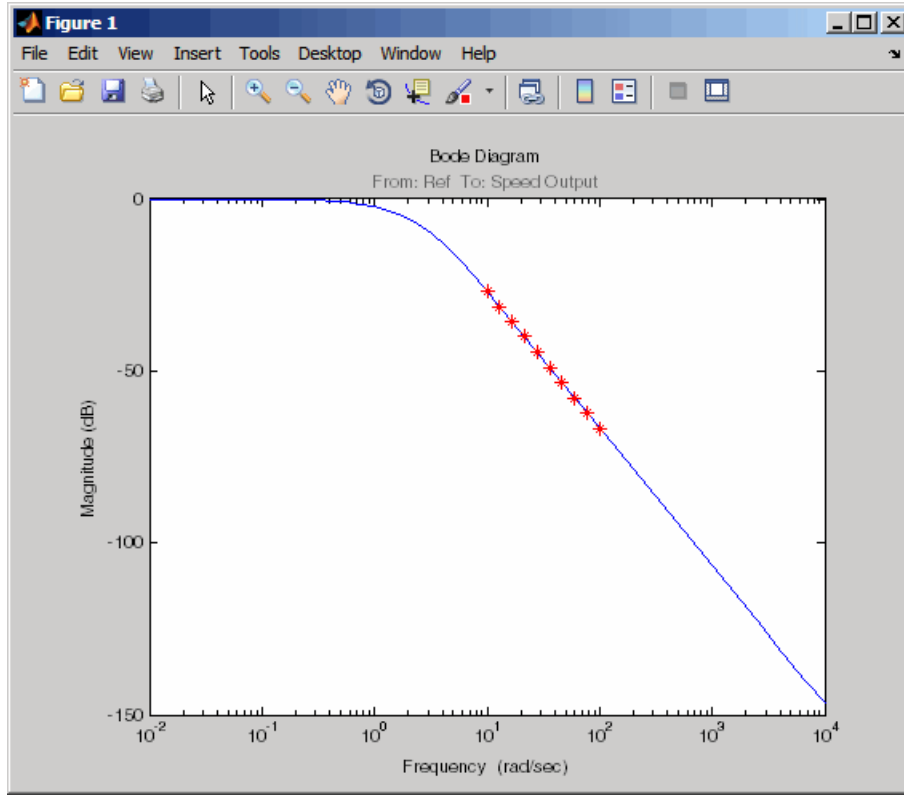
```
ans =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    'scdspeed_ctrlloop/Engine Model'
      'scdspeed_plantref/Drag Torque/Step1'
```

Now you can estimate the frequency response without the contribution of the time-varying source blocks. To do so, set the `BlocksToHoldConstant` option of `frestimateOptions` equal to `srcblks`, and run the estimation.

```
opts = frestimateOptions
opts.BlocksToHoldConstant = srcblks
% Run frestimate again with blocks disabled
 [sysest2,simout2] = frestimate(mdl,io,in,opts);
```

The frequency response estimate now provides a good match to the exact linearization result.

```
bodemag(sys,sysest2,'r*')
```



Tips

- Use `frest.findSources` to identify time-varying source blocks that can interfere with frequency response estimation. To disable such blocks to estimate frequency response, set the `BlocksToHoldConstant` option of `frestimateOptions` equal to `blocks` or a subset of `blocks`. Then, estimate the frequency response using `frestimate`.
- Sometimes, `model` includes referenced models containing source blocks in the signal path of a linearization output point. In such cases, set the referenced models to normal simulation mode to ensure that `frest.findSources` locates them. Use the `set_param` command to set `SimulationMode` of any referenced models to `Normal` before running `frest.FindSources`.

Algorithms

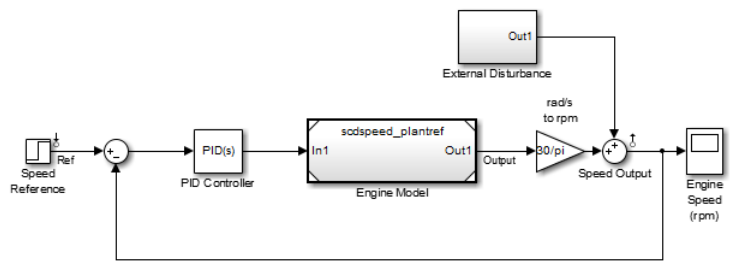
To locate time-varying source blocks that can interfere with frequency response estimation, `frest.findSources` begins at each linearization output point in the model. From each output point, the algorithm traces every signal path backward block by block. The algorithm reports any source block (a block with no input port) it discovers, unless that source block is a Constant or Ground block.

The `frest.findSources` algorithm traces every signal path that can affect the signal value at each linearization output point in the model. The paths traced include:

- Signal paths inside virtual and nonvirtual subsystems.
- Signal paths inside normal-mode referenced models. Set all referenced models to normal simulation mode before using `frest.findSources` to ensure that the algorithm identifies source blocks within the referenced models.
- Signals routed through From and Goto blocks, or through Data Store Read and Data Store Write blocks.
- Signals routed through switches. The `frest.findSources` algorithm assumes that any pole of a switch can be active during frequency response estimation. The algorithm therefore follows the signal back through all switch inputs.

For example, consider the model `scdspeed_ctrlloop`. This model has one linearization output point, located at the output of the Sum block labeled `Speed Output`. (The `frest.findSources` algorithm ignores linearization input points.) Before running `frest.findSources`, convert the referenced model to normal simulation mode:

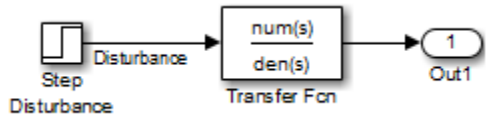
```
set_param('scdspeed_ctrlloop/Engine Model',...
                        'SimulationMode','Normal');
```



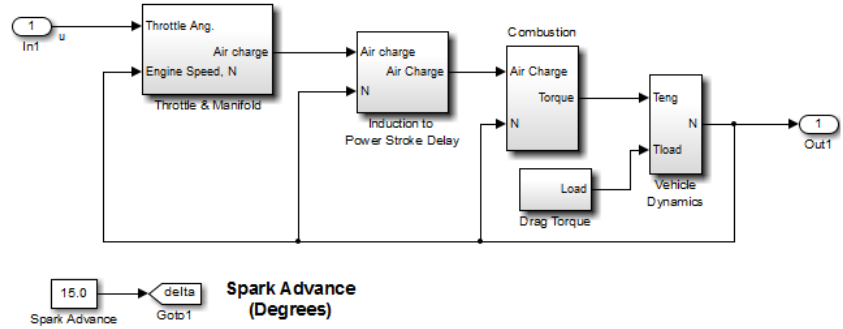
You can now run `frest.findSources` to identify the time-varying source blocks using the linearization output point defined in the model.

```
srcblks = frest.findSources('scdspeed_ctrlloop');
```

The algorithm begins at the output point and traces back through the Sum block `Speed Output`. One input to `Speed Output` is the subsystem `External Disturbance`. The algorithm enters the subsystem, finds the source block labeled `Step Disturbance`, and reports that block.



The Sum block `Speed Output` has another input, which the algorithm traces back into the referenced model `Engine Model`. `Engine Model` contains several subsystems, and the algorithm traces the signal through these subsystems to identify any time-varying source blocks present.



For example, the `Combustion` subsystem includes the From block marked `delta` that routes the signal from the `Spark Advance` source. Because `Spark Advance` is a constant source block, however, the algorithm does not report the presence of the block.

The algorithm continues the trace until all possible signal paths contributing to the signal at each linearization output point are examined.

Alternatives

You can use the Simulink Model Advisor to determine whether time-varying source blocks exist in the signal path of output linear analysis points in your model. To do so,

use the Model Advisor check “Identify time-varying source blocks interfering with frequency response estimation.” For more information about using the Model Advisor, see “Run Model Checks” (Simulink) in the *Simulink User's Guide*.

See Also

`frestimate` | `frestimateOptions`

Topics

“Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-58

Introduced in R2010b

frest.Random

Package: frest

Random input signal for simulation

Syntax

```
input = frest.Random('OptionName',OptionValue)
input = frest.Random(sys)
```

Description

`input = frest.Random('OptionName',OptionValue)` creates the Random input signal using the options specified by comma-separated name/value pairs.

`input = frest.Random(sys)` creates a Random input signal based on the dynamics of a linear system *`sys`*.

To view a plot of your input signal, type `plot(input)`. To obtain a time series for your input signal, use the `generateTimeseries` command.

Input Arguments

`sys`

Linear system for creating a random signal based on the dynamic characteristics of this system. You can specify the linear system based on known dynamics using `tf`, `zpk`, or `ss`. You can also obtain the linear system by linearizing a nonlinear system.

The resulting random signal automatically sets these options based on the linear system:

- `Ts` is set such that the Nyquist frequency of the signal is five times the upper end of the frequency range to avoid aliasing issues.

- `NumSamples` is set such that the frequency response estimation includes the lower end of the frequency range.

Other random options have default values.

`'OptionName',OptionValue`

Signal characteristics, specified as comma-separated option name and option value pairs.

Option Name	Option Value
`'Amplitude'`	Signal amplitude. **Default**: `1e-5`
`'Ts'`	Sample time of the chirp signal in seconds. **Default**: `1e-3`
`'NumSamples'`	Number of samples in the Random signal. **Default**: `1e4`
`'Stream'`	Random number stream you create using the MATLAB command `RandStream`. The state of the `stream` you specify stores with the input signal. This stored state allows the software to return the same result every time you use `generateTimeseries` and `frestimate` with the input signal. **Default**: Default stream of the MATLAB session

Examples

Create a Random input signal with 1000 samples taken at 100 Hz and amplitude of 0.02:

```
input = frest.Random('Amplitude',0.02,'Ts',1/100,'NumSamples',1000);
```

Create a Random input signal using multiplicative lagged Fibonacci generator random stream:

```
% Specify the random number stream
stream = RandStream('mlfg6331_64','Seed',0);

% Create the input signal
input = frest.Random('Stream',stream);
```

See Also

`frest.Random` | `frest.Sinestream` | `frestimate` | `generateTimeseries` | `getSimulationTime`

Topics

"Estimation Input Signals" on page 5-7
"Estimate Frequency Response at the Command Line" on page 5-33
"Estimate Frequency Response Using Linear Analysis Tool" on page 5-26

Introduced in R2009b

frest.simCompare

Package: frest

Plot time-domain simulation of nonlinear and linear models

Syntax

```
frest.simCompare(simout,sys,input)
frest.simCompare(simout,sys,input,x0)
[y,t] = frest.simCompare(simout,sys,input)
[y,t,x] = frest.simCompare(simout,sys,input,x0)
```

Description

`frest.simCompare(simout,sys,input)` plots both

- Simulation output, `simout`, of the nonlinear Simulink model

  You obtain the output from the `frestimate` command.
- Simulation output of the linear model `sys` for the input signal `input`

  The linear simulation results are offset by the initial output values in the `simout` data.

`frest.simCompare(simout,sys,input,x0)` plots the frequency response simulation output and the simulation output of the linear model with initial state `x0`. Because you specify the initial state, the linear simulation result is *not* offset by the initial output values in the `simout` data.

`[y,t] = frest.simCompare(simout,sys,input)` returns the linear simulation output response `y` and the time vector `t` for the linear model `sys` with the input signal `input`. This syntax does not display a plot. The matrix `y` has as many rows as time samples `(length(t))` and as many columns as system outputs.

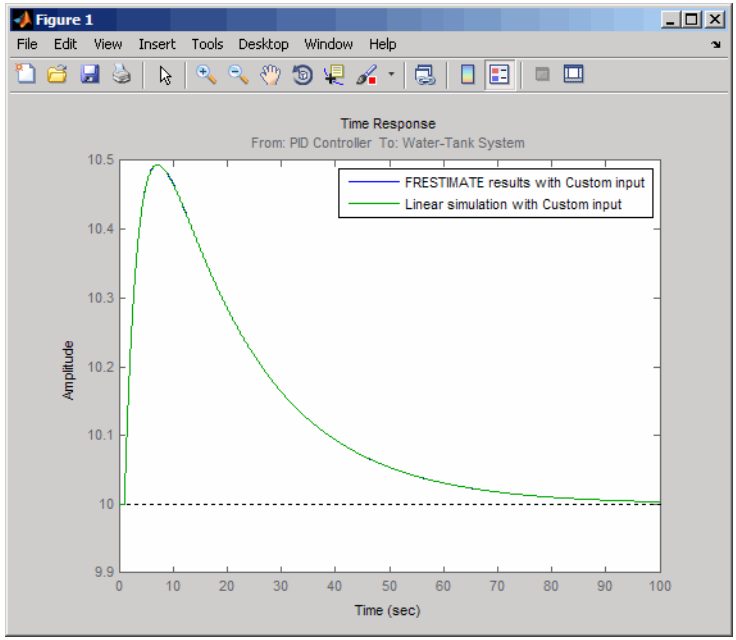
`[y,t,x] = frest.simCompare(simout,sys,input,x0)` also returns the state trajectory `x` for the linear state space model `sys` with initial state `x0`.

Examples

Compare a time-domain simulation of the Simulink `watertank` model and its linear model representation:

```
% Create input signal for simulation
input = frest.createStep('FinalTime',100);

% Open the Simulink model
watertank

% Specify the operating point for the estimation
watertank_spec = operspec('watertank');
op = findop('watertank',watertank_spec)

% Specify portion of model to estimate
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'output');

% Estimate the frequency response of the watertank model
[sysest,simout] = frestimate('watertank',op,io,input)
sys = linearize('watertank',op,io);
frest.simCompare(simout,sys,input);
```

The software returns the following plot.



See Also

`frest.simView` | `frestimate`

Introduced in R2009b

frest.simView

Package: frest

Plot frequency response model in time- and frequency-domain

Syntax

```
frest.simView(simout,input,sysest)
frest.simView(simout,input,sysest,sys)
```

Description

`frest.simView(simout,input,sysest)` plots the following frequency response estimation results:

- Time-domain simulation `simout` of the Simulink model
- FFT of time-domain simulation `simout`
- Bode of estimated system `sysest`

  This Bode plot is available when you create the input signal using `frest.Sinestream` or `frest.Chirp`. In this plot, you can interactively select frequencies or a frequency range for viewing the results in all three plots.

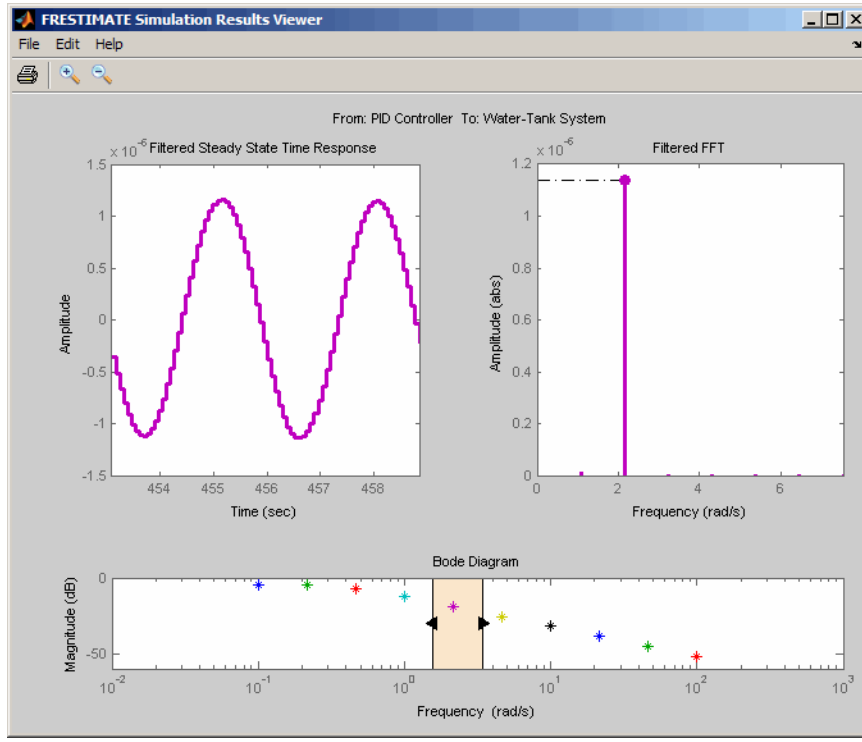
You obtain `simout` and `sysest` from the `frestimate` command using the input signal `input`.

`frest.simView(simout,input,sysest,sys)` includes the linear system `sys` in the Bode plot when you create the input signal using `frest.Sinestream` or `frest.Chirp`. Use this syntax to compare the linear system to the frequency response estimation results.

Examples

Estimate the closed-loop of the `watertank` Simulink model and analyze the results:

```
% Open the Simulink model
watertank

% Specify portion of model to linearize and estimate
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'output');

% Specify the operating point for the linearization and estimation
watertank_spec = operspec('watertank');
op = findop('watertank',watertank_spec);

% Create input signal for simulation
input = frest.Sinestream('Frequency',logspace(-1,2,10));

% Estimate the frequency response of the watertank model
[sysest,simout] = frestimate('watertank',op,io,input);

% Analyze the estimation results
frest.simView(simout,input,sysest)
```



See Also

`frest.simCompare` | `frestimate`

Topics

"Analyze Estimated Frequency Response" on page 5-38
"Troubleshooting Frequency Response Estimation" on page 5-46

Introduced in R2009b

frest.Sinestream

Package: frest

Signal containing series of sine waves

Syntax

```
input = frest.Sinestream(sys)
input = frest.Sinestream('OptionName',OptionValue)
```

Description

`input = frest.Sinestream(sys)` creates a signal with a series of sinusoids based on the dynamics of a linear system *`sys`*.

`input = frest.Sinestream('OptionName',OptionValue)` creates a signal with a series of sinusoids, where each sinusoid frequency lasts for a specified number of periods, using the options specified by comma-separated name/value pairs.

To view a plot of your input signal, type `plot(input)`. To obtain a timeseries for your input signal, use the `generateTimeseries` command.

Input Arguments

`sys`

Linear system for creating a sinestream signal based on the dynamic characteristics of this system. You can specify the linear system based on known dynamics using `tf`, `zpk`, or `ss`. You can also obtain the linear system by linearizing a nonlinear system.

The resulting sinestream signal automatically sets these options based on the linear system:

- `'Frequency'` are the frequencies at which the linear system has interesting dynamics.

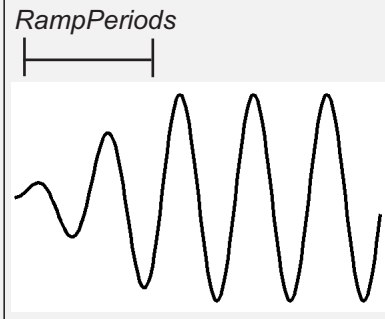
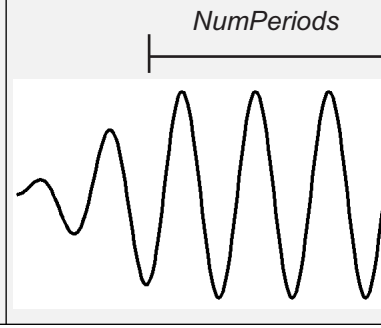
- `'SettlingPeriods'` is the number of periods it takes the system to reach steady state at each frequency in `'Frequency'`.
- `'NumPeriods'` is (3 + `SettlingPeriods`) to ensure that each frequency excites the system at specified amplitude for at least three periods.
- For discrete systems only, `'SamplesPerPeriod'` is set such that all frequencies have the same sample time as the linear system.

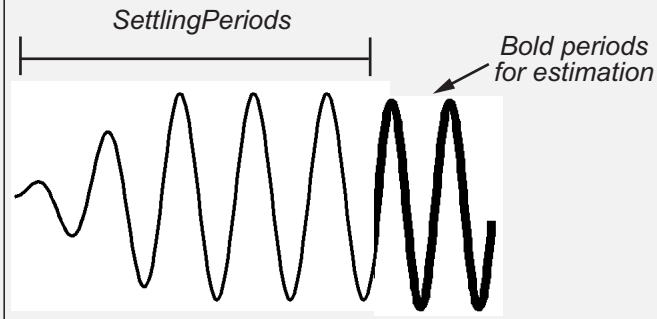
Other sinestream options have default values.

`'OptionName',OptionValue`

Signal characteristics, specified as comma-separated option name and option value pairs.

Option Name	Option Value
`'Frequency'`	Signal frequencies, specified as either a scalar or a vector of frequency values. **Default**: `logspace(1,3,50)`
`'Amplitude'`	Signal amplitude at each frequency, specified as either: • Scalar to set all frequencies to same value • Vector to set each frequencies to a different value **Default**: `1e-5`
`'SamplesPerPeriod'`	Number of samples for each period for each signal frequency, specified as either: • Scalar to set all frequencies to same value • Vector to set each frequencies to a different value **Default**: `40`
`'FreqUnits'`	Frequency units: • `'rad/s'` — Radians per second • `'Hz'` — Hertz **Default**: `'rad/s'`

Option Name	Option Value
`'RampPeriods'`	Number of periods for ramping up the amplitude of each sine wave to its maximum value, specified as either: • Scalar to set all frequencies to same value • Vector to set each frequencies to a different value Use this option to ensure a smooth response when your input amplitude changes. **Default**: `0` 
`'NumPeriods'`	Number of periods each sine wave is at maximum amplitude, specified as either: • Scalar to set all frequencies to same value • Vector to set each frequencies to a different value **Default**: `max(3 - RampPeriods + SettlingPeriods, 2)` 

Option Name	Option Value
`'SettlingPeriods'`	Number of periods corresponding to the transient portion of the simulated response at a specific frequency, before the system reaches steady state, specified as either: • Scalar to set all frequencies to same value • Vector to set each frequencies to a different value Before performing the estimation, `frestimate` discards this number of periods from the output signals. **Default**: `1` 
`'ApplyFilteringInFRESTIMATE'`	Frequency-selective FIR filtering of the input signal before estimating the frequency response using `frestimate`. • `'on'` (default) • `'off'` For more information, see the `frestimate` algorithm.

Option Name	Option Value
`'SimulationOrder'`	The order in which `frestimate` injects the individual frequencies of the input signal into your Simulink model during simulation. • `'Sequential'` (default) — `frestimate` injects one frequency after the next into your model in a single Simulink simulation using variable sample time. To use this option, your Simulink model must use a variable-step solver. • `'OneAtATime'` — `frestimate` injects each frequency during a separate Simulink simulation of your model. Before each simulation, `frestimate` initializes your Simulink model to the operating point specified for estimation. If you have Parallel Computing Toolbox installed, you can run each simulation in parallel to speed up estimation using parallel computing. For more information, see “Speeding Up Estimation Using Parallel Computing” on page 5-80.

Examples

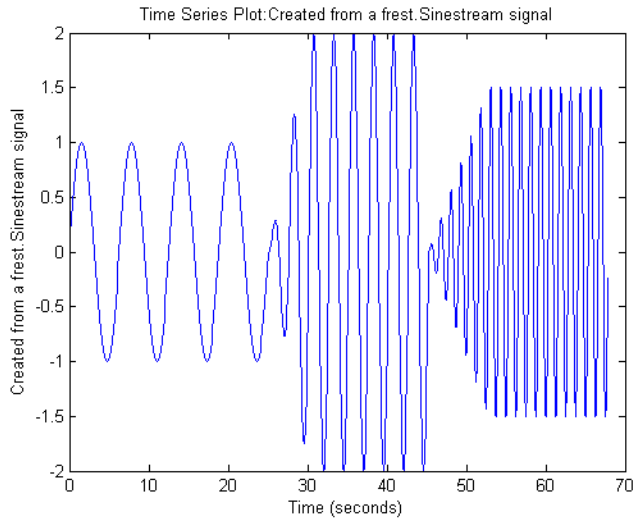
Create a sinestream signal having several different frequencies. For each frequency, specify an amplitude, a number of periods at maximum amplitude, a ramp-up period, and a number of settling periods.

1 Create sinestream signal.

```
input = frest.Sinestream('Frequency',[1 2.5 5],...
             'Amplitude',[1 2 1.5],...
             'NumPeriods',[4 6 12],...
             'RampPeriods',[0 2 6],...
             'SettlingPeriods',[1 3 7]);
```

2 (Optional) Plot the sinestream signal.

```
plot(input)
```



Create a sinusoidal input signal with the following characteristics:

- 50 frequencies spaced logarithmically between 10 Hz and 1000 Hz
- All frequencies have amplitude of 1e-3
- Sampled with a frequency 10 times the frequency of the signal (meaning ten samples per period)

```
% Create the input signal
input = frest.Sinestream('Amplitude',1e-3,'Frequency',logspace(1,3,50),...
'SamplesPerPeriod',10,'FreqUnits','Hz');
```

See Also

`frest.Chirp` | `frest.Random` | `frest.createFixedTsSinestream` | `frestimate` | `generateTimeseries` | `getSimulationTime`

Topics

“Estimation Input Signals” on page 5-7
“Estimate Frequency Response at the Command Line” on page 5-33
“Estimate Frequency Response Using Linear Analysis Tool” on page 5-26
“Speeding Up Estimation Using Parallel Computing” on page 5-80

Introduced in R2009b

frestimate

Frequency response estimation of Simulink models

Syntax

```
sysest = frestimate(model,io,input)
sysest = frestimate(model,op,io,input)
[sysest,simout] = frestimate(model,op,io,input)
[sysest,simout] = frestimate(model,op,io,input,options)
```

Description

`sysest = frestimate(model,io,input)` estimates frequency response model `sysest`. `model` is a character vector or string that specifies the name of your Simulink model. `input` can be a sinestream, chirp, or random signal, or a MATLAB timeseries object. `io` specifies the linearization I/O object, which you either obtain using `getlinio` or create using `linio`. I/O points cannot be on bus signals. The estimation occurs at the operating point specified in the Simulink model.

`sysest = frestimate(model,op,io,input)` initializes the model at the operating point `op` before estimating the frequency response model. Create `op` using either `operpoint` or `findop`.

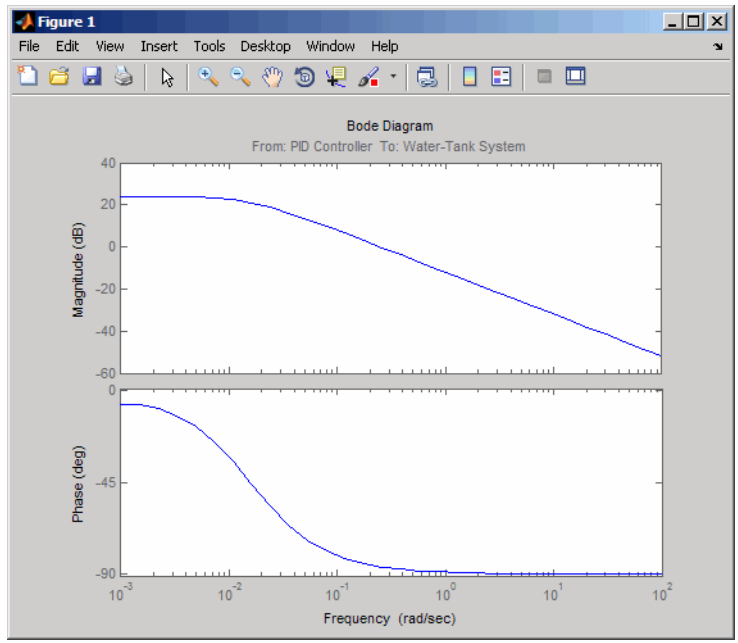
`[sysest,simout] = frestimate(model,op,io,input)` estimates frequency response model and returns the simulated output `simout`. This output is a cell array of `Simulink.Timeseries` objects with dimensions `m`-by-`n`. `m` is the number of linearization output points, and `n` is the number of input channels.

`[sysest,simout] = frestimate(model,op,io,input,options)` uses the frequency response options (`options`) to estimate the frequency response. Specify these options using `frestimateOptions`.

Examples

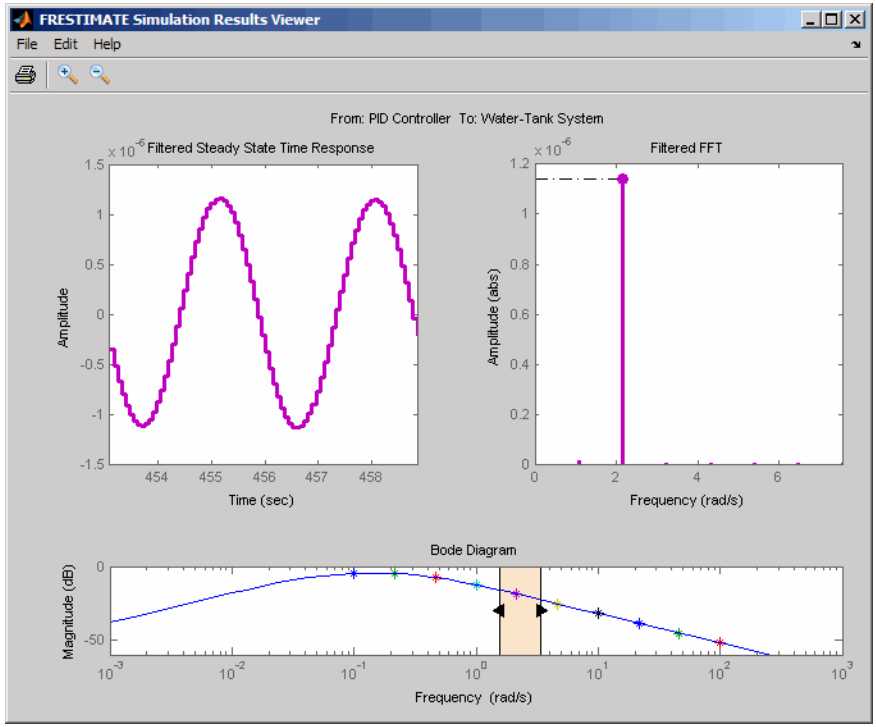
Estimating frequency response for a Simulink model:

```
% Create input signal for simulation:
input = frest.Sinestream('Frequency',logspace(-3,2,30));

% Open the Simulink model:
watertank

% Specify portion of model to estimate:
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

% Specify the steady state operating point for the estimation.
watertank_spec = operspec('watertank');
op = findop('watertank',watertank_spec);

% Estimate frequency response of specified blocks:
sysest = frestimate('watertank',op,io,input);
bode(sysest)
```



Validate exact linearization results using estimated frequency response of a Simulink model:

```
% Open the Simulink model:
watertank

% Specify portion of model to estimate:
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'output');

% Specify operating point for linearization and estimation:
watertank_spec = operspec('watertank');
op = findop('watertank',watertank_spec);

% Linearize the model:
sys = linearize('watertank',op,io);

% Estimate the frequency response of the watertank model
input = frest.Sinestream('Frequency',logspace(-1,2,10));
[sysest,simout] = frestimate('watertank',op,io,input);

% Compare linearization and estimation results in frequency domain:
frest.simView(simout,input,sysest,sys)
```

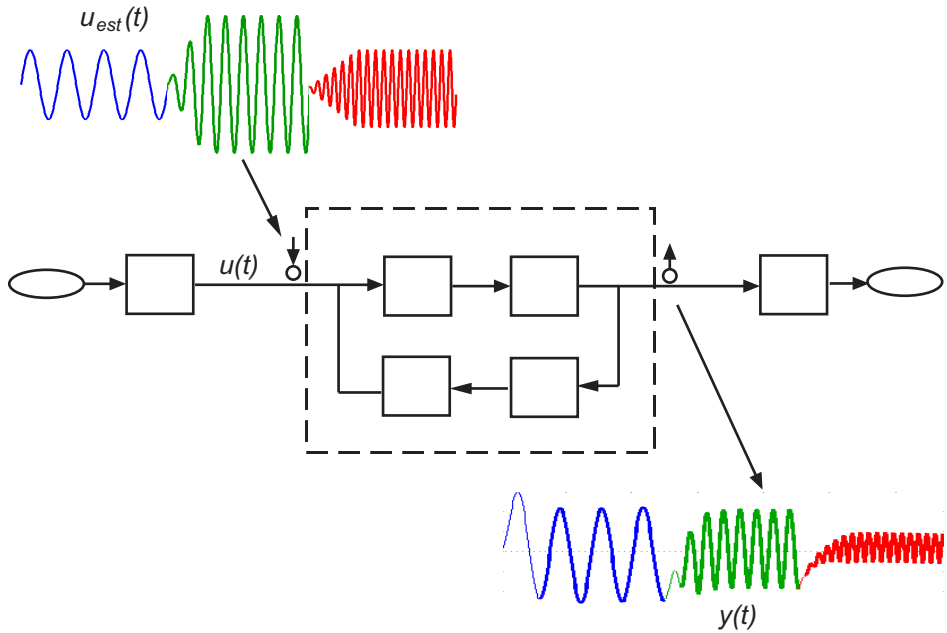


Algorithms

`frestimate` performs the following operations when you use the sinestream signal:

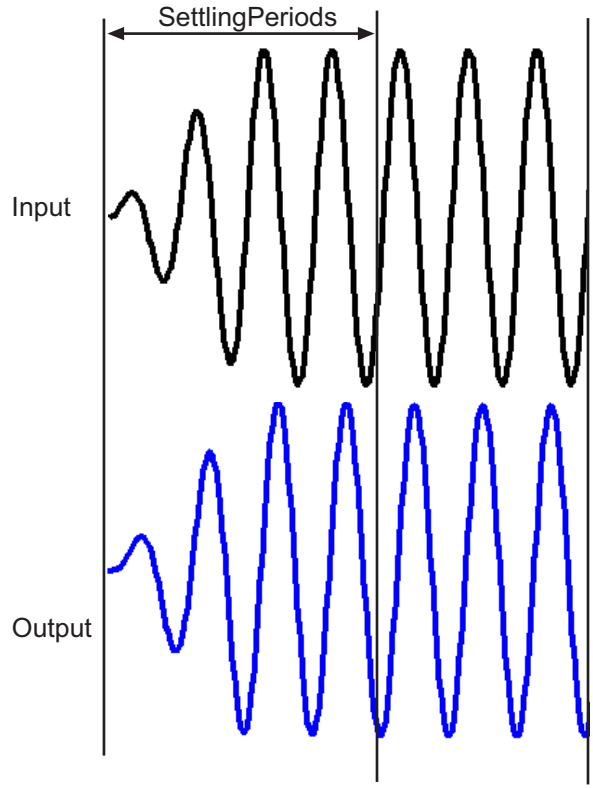
1. Injects the sinestream input signal you design, $u_{est}(t)$, at the linearization input point.
2. Simulates the output at the linearization output point.

   `frestimate` adds the signal you design to existing Simulink signals at the linearization input point.



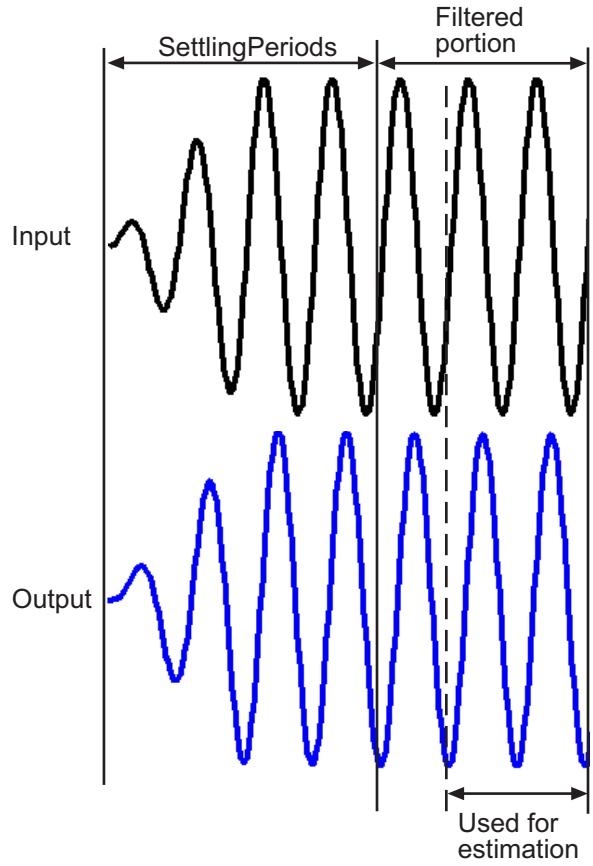
3 Discards the `SettlingPeriods` portion of the output (and the corresponding input) at each frequency.

The simulated output at each frequency has a transient portion and steady state portion. `SettlingPeriods` corresponds to the transient components of the output and input signals. The periods following `SettlingPeriods` are considered to be at steady state.



4 Filters the remaining portion of the output and the corresponding input signals at each input frequency using a bandpass filter. Because most models are not at steady state, the response might contain low-frequency transient behavior. Filtering typically improves the accuracy of your model by removing the effects of frequencies other than the input frequencies, which are problematic when sampling and analyzing data of finite length. These effects are called *spectral leakage*.

Any transients associated with filtering are only in the first period of the filtered steady-state output. After filtering, `frestimate` discards the first period of the input and output signals. `frestimate` uses a finite impulse response (FIR) filter, whose order matches the number of samples in a period.



5 Estimates the frequency response of the processed signal by computing the ratio of the fast Fourier transform of the filtered steady-state portion of the output signal $y_{est}(t)$ and the fast Fourier transform of the filtered input signal $u_{est}(t)$:

$$Frequency\ Response\ Model = \frac{fft\ of\ y_{est}(t)}{fft\ of\ u_{est}(t)}$$

To compute the response at each frequency, `frestimate` uses only the simulation output at that frequency.

See Also

`frest.Chirp` | `frest.Random` | `frest.Sinestream` | `frest.simView` | `frestimateOptions` | `getSimulationTime`

Topics

“Estimate Frequency Response at the Command Line” on page 5-33
“Estimate Frequency Response Using Linear Analysis Tool” on page 5-26
“Speeding Up Estimation Using Parallel Computing” on page 5-80

Introduced in R2009b

frestimateOptions

Options for frequency response estimation

Syntax

```
options = frestimateOptions
options = frestimateOptions('OptionName',OptionValue)
```

Description

`options = frestimateOptions` creates a frequency response estimation options object, `options`, with default settings. Pass this object to the function `frestimate` to use these options for frequency response estimation.

`options = frestimateOptions('OptionName',OptionValue)` creates a frequency response estimation options object `options` using the options specified by comma-separated name/value pairs.

Input Arguments

`'OptionName',OptionValue`

Estimation options, specified as comma-separated option name and option value pairs.

Option Name	Option Value
`'BlocksToHoldConstant'`	Block paths of time-varying source blocks to hold constant during frequency response estimation. specified as a cell array of character vectors or a string array. Use `frest.findSources` to identify time-varying source blocks that can interfere with frequency response estimation. **Default**: empty

Option Name	Option Value
`'UseParallel'`	Set to `'on'` to enable parallel computing for estimations with the `frestimate` command. **Default**: `'off'`
`'ParallelPathDependencies'`	A cell array of character vectors or string array that specifies the path dependencies required to execute the model to estimate. All the workers in the parallel pool must have access to the folders listed in `'ParallelPathDependencies'`. **Default**: empty

Examples

Identify and disable time-varying source blocks for frequency response estimation.

```
% Open Simulink model.
mdl = 'scdspeed_ctrlloop';
open_system(mdl)

% Convert referenced subsystem to normal mode.
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal');

% Get I/O points and create sinestream.
io = getlinio(mdl)
in = frest.Sinestream('Frequency',logspace(1,2,10),'NumPeriods',30,...
              'SettlingPeriods',25);

% Identify time-varying source blocks.
srcblks = frest.findSources(mdl)

% Create options set specifying blocks to hold constant
opts = frestimateOptions
opts.BlocksToHoldConstant = srcblks

% Run frestimate
[sysest,simout] = frestimate(mdl,io,in,opts)
```

Enable parallel computing and specify the model path dependencies.

```
% Copy referenced model to temporary folder.
pathToLib = scdpathdep_setup;

% Add folder to search path.
addpath(pathToLib);

% Open Simulink model.
mdl = 'scdpathdep';
open_system(mdl);

% Get model dependency paths.
dirs = frest.findDepend(mdl)

% The resulting path is on a local drive, C:/.
% Replace C:/ with valid network path accessible to remote workers.
dirs = regexprep(dirs,'C:/','\\\\hostname\\C$\\')

% Enable parallel computing and specify the model path dependencies.
options = frestimateOptions('UseParallel','on','ParallelPathDependencies',dirs)
```

Alternatives

You can enable parallel computing for all models with no path dependencies. To do so, select the **Use the parallel pool when you use the "frestimate" command** check box in the MATLAB preferences. When you select this check box and use the `frestimate` command, you do not need to provide a frequency response options object.

If your model has path dependencies, you must create your own frequency response options object that specifies the path dependencies. Use the `ParallelPathDependencies` option before beginning the estimation.

See Also

`frest.findSources` | `frestimate`

Introduced in R2010a

fselect

Extract sinestream signal at specified frequencies

Syntax

```
input2 = fselect(input,fmin,fmax)
input2 = fselect(input,index)
```

Description

`input2 = fselect(input,fmin,fmax)` extracts a portion of the sinestream input signal `input` in the frequency range between `fmin` and `fmax`. Specify `fmin` and `fmax` in the same frequency units as the sinestream signal.

`input2 = fselect(input,index)` extracts a sinestream signal at specific frequencies, specified by the vector of indices `index`.

Examples

Extract the second frequency in a sinestream signal:

```
% Create the input signal
input = frest.Sinestream('Frequency',[1 2.5 5],...
                         'Amplitude',[1 2 1.5],...
                         'NumPeriods',[4 6 12],...
                         'RampPeriods',[0 2 6]);

% Extract a sinestream signal for the second frequency
input2 = fselect(input,2)

% Plot the extracted input signal
plot(input2)
```

See Also

`fdel` | `frest.Sinestream` | `frestimate`

Topics

“Time Response Not at Steady State” on page 5-46

Introduced in R2010a

generateTimeseries

Generate time-domain data for input signal

Syntax

```
ts = generateTimeseries(input)
```

Description

`ts = generateTimeseries(input)` creates a MATLAB `timeseries` object `ts` from the input signal `input`. `input` can be a sinestream, chirp, or random signal. For chirp and random signals, that time vector of `ts` has equally spaced time values, ranging from 0 to `Ts(NumSamples-1)`.

Examples

Create `timeseries` object for chirp signal:

```
input = frest.Chirp('Amplitude',1e-3,'FreqRange',...
                    [10 500],'NumSamples',20000);
ts = generateTimeseries(input)
```

See Also

`frest.Chirp` | `frest.Random` | `frest.Sinestream` | `frestimate`

Introduced in R2009b

get

Properties of linearization I/Os and operating points

Syntax

```
get(ob)
get(ob,'PropertyName')
```

Description

`get(ob)` displays all properties and corresponding values of the object, `ob`, which can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`get(ob,'PropertyName')` returns the value of the property, `PropertyName`, within the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`ob.PropertyName` is an alternative notation for displaying the value of the property, `PropertyName`, of the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

Examples

Create an operating point object, `op`, for the Simulink model, `magball`.

```
op=operpoint('magball');
```

Get a list of all object properties using the `get` function with the object name as the only input.

```
get(op)
```

This returns the properties of `op` and their current values.

```
     Model: 'magball'
     States: [5x1 opcond.StatePoint]
     Inputs: [0x1 double]
       Time: 0
    Version: 2
```

To view the value of a particular property of `op`, supply the property name as an argument to `get`. For example, to view the name of the model associated with the operating point object, type:

```
V=get(op,'Model')
```

which returns

```
V =
magball
```

Because `op` is a structure, you can also view any properties or fields using dot-notation, as in this example.

```
W=op.States
```

This notation returns a vector of objects containing information about the states in the operating point.

```
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
```

Use `get` to view details of `W`. For example:

```
get(W(2),'x')
```

returns

```
ans =

   14.0071
```

See Also

findop | getlinio | linio | operpoint | operspec | set

Introduced before R2006a

getBlockInfo

Package: linearize.advisor

Obtain diagnostic information for block linearizations

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. You can troubleshoot your linearization results by reviewing this diagnostic information. To access the diagnostic information, use the `getBlockInfo` function.

Syntax

```
blockInfo = getBlockInfo(advisor)
blockInfo = getBlockInfo(advisor,block)
blockInfo = getBlockInfo(advisor,index)
```

Description

`blockInfo = getBlockInfo(advisor)` returns the diagnostic information for all blocks listed in the `LinearizationAdvisor` object, `advisor`.

`blockInfo = getBlockInfo(advisor,block)` returns diagnostic information for blocks with block paths specified in `block`.

`blockInfo = getBlockInfo(advisor,index)` returns diagnostic information for blocks with indices specified in `index`.

Examples

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)
```

```
diags =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Obtain Diagnostics Using Block Names

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the saturation block.

```
satDiag = getBlockInfo(advisor,'scdpendulum/pendulum/Saturation')

satDiag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostic information for multiple blocks at once. Obtain diagnostics for the sin blocks in the model.

```
sinBlocks = {'scdpendulum/pendulum/Trigonometric Function';
             'scdpendulum/angle_wrap/Trigonometric Function1'};

sinDiag = getBlockInfo(advisor,sinBlocks)

sinDiag =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Obtain Diagnostics Using Indices

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the first element of `advisor.BlockDiagnostics`.

```
diag = getBlockInfo(advisor,1)
```

```
diag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostics for multiple blocks. For example, obtain diagnostics for the second and third blocks listed in `advisor`.

```
diags = getBlockInfo(advisor,[2 3])
```

```
diags =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Obtain Diagnostics for Blocks in Subsystem

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain block paths of linearized blocks.

```
paths = getBlockPaths(advisor);
```

Create boolean array indicating which blocks are in the `angle_wrap` subsystem.

```
index = contains(paths,'angle_wrap');
```

Obtain diagnostic information for these blocks.

```
diags = getBlockInfo(advisor,index)
```

```
diags =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

`block` — Block paths

character vector | cell array of character vectors

Block paths in Simulink model, specified as one of the following:

- Character vector — Obtain diagnostic information for a single block.
- Cell array of character vectors — Obtain diagnostic information for multiple blocks.

`index` — Block indices

positive integer | array of positive integers | boolean array

Block indices, specified as one of the following:

- Positive integer — Obtain diagnostic information for the specified element of `Advisor.BlockDiagnostics`
- Array of positive integers — Obtain diagnostic information for multiple elements of `Advisor.BlockDiagnostics`.
- Boolean array — For each element of `index` that is `true`, return the diagnostics for the corresponding element of `Advisor.BlockDiagnostics`.

Output Arguments

`blockInfo` — Diagnostic information for block linearizations

`BlockDiagnostic` object | vector of `BlockDiagnostic` objects | cell array

Diagnostic information for block linearizations indicated by `index`, returned as a `BlockDiagnostic` object or vector of `BlockDiagnostic` objects if `advisor` is a single `LinearizationAdvisor` object.

If `advisor` is an array of `LinearizationAdvisor` objects, then `blockInfo` is a cell array with the same dimensions as `advisor` in which each element is a vector of `BlockDiagnostic` objects.

See Also

Using Objects

`LinearizationAdvisor`

Functions

`advise` | `find` | `getBlockPaths`

Topics

“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

getBlockPaths

Package: linearize.advisor

Obtain list of blocks in `LinearizationAdvisor` object

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations, which you can use for troubleshooting linearization results. To obtain a list of the blocks in the `LinearizationAdvisor` object, use the `getBlockPaths` function.

Syntax

```
blocks = getBlockPaths(advisor)
```

Description

`blocks = getBlockPaths(advisor)` returns a list of block paths for the blocks in the `LinearizationAdvisor` object `advisor`.

Examples

Obtain List of Numerically Perturbed Blocks

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Find all blocks in linearization results that are numerically perturbed.

```
perturbed = find(advisor,linqueryIsNumericallyPerturbed);
```

Obtain list of numerically perturbed blocks.

```
blocks = getBlockPaths(perturbed)
```

```
blocks = 6x1 cell array
    {'scdspeed/Throttle & Manifold/Intake Manifold/Convert to mass charge'}
    {'scdspeed/Combustion/Torque Gen'                                      }
    {'scdspeed/Combustion/Torque Gen2'                                     }
    {'scdspeed/Throttle & Manifold/Intake Manifold/Pumping1'               }
    {'scdspeed/Throttle & Manifold/Throttle/f(theta)'                      }
    {'scdspeed/Throttle & Manifold/Throttle/g(pratio)'                     }
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

Output Arguments

`blocks` — Block paths

cell array of character vectors | cell array

Block paths for blocks in `advisor`, returned as a cell array of character vectors if `advisor` is a single `LinearizationAdvisor` object. If `advisor` is an array of `LinearizationAdvisor` objects, then `blocks` is a cell array with the same dimensions as `advisor` in which each element is a cell array of character vectors.

See Also

Using Objects

`LinearizationAdvisor`

Functions

`advise` | `getBlockInfo` | `getBlockPaths`

Topics

“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

getInputIndex

Get index of an input element of an operating point specification or operating point

The `Inputs` property of an operating point specification is an array that contains trimming specifications for each model input. When defining a mapping function for customized trimming of Simulink models, `getInputIndex` lets you obtain the index of an input specification based on the corresponding block path.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.

Syntax

```
index = getInputIndex(op,block)
index = getInputIndex(op,block,element)
```

Description

`index = getInputIndex(op,block)` returns the index of the input specification that corresponds to `block` in the `Inputs` property of operating point specification `op`.

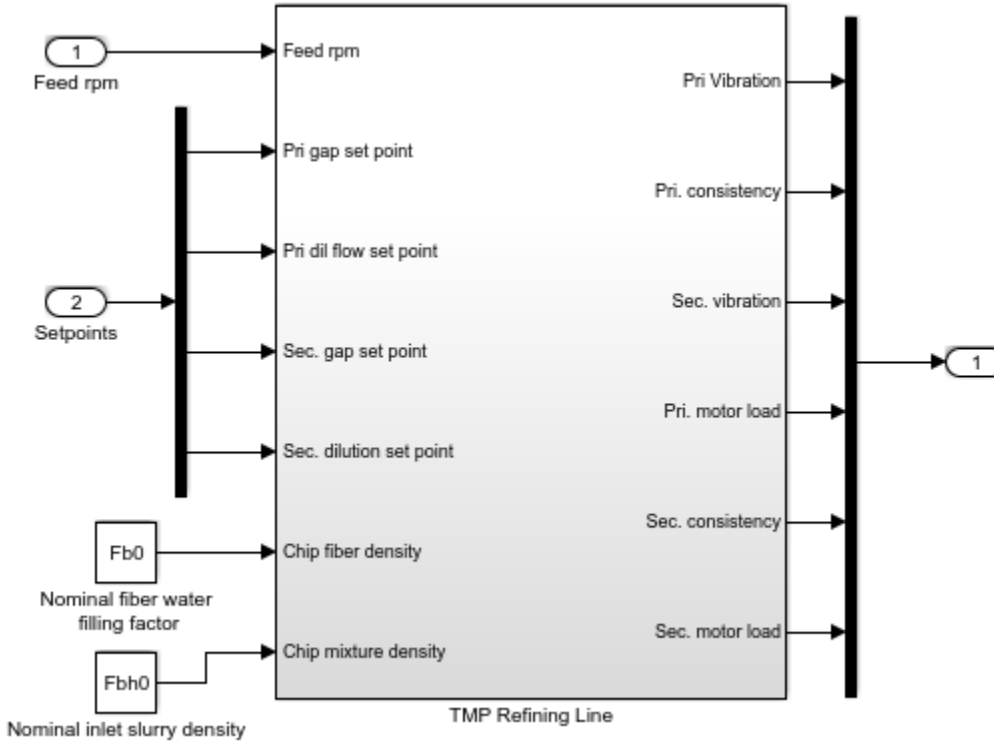
`index = getInputIndex(op,block,element)` returns the index of the specified `element` within an input specification for an input port that has a port width greater than `1`.

Examples

Get Input Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdtmpSetpoints';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains specifications for the root-level input ports of the model.

```
opspec.Inputs
```

```
(1.) scdtmpSetpoints/Feed rpm
     initial guess: 0
```

```
(2.) scdtmpSetpoints/Setpoints
     initial guess: 0
     initial guess: 0
     initial guess: 0
     initial guess: 0
```

Obtain the index of the specification in `opspec.Inputs` that corresponds to the Feed rpm input block.

```
index1 = getInputIndex(opspec,'scdtmpSetpoints/Feed rpm')
```

```
index1 =

     1     1
```

`index1(1)` is the index of the input specification object for the Feed rpm block in the `opspec.Inputs`. Since this input port is a scalar signal, `index1` has one row and `index1(2)` is `1`.

If an input port is a vector signal, you can obtain the indices for all of the elements in the corresponding input specification.

```
index2 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints')
```

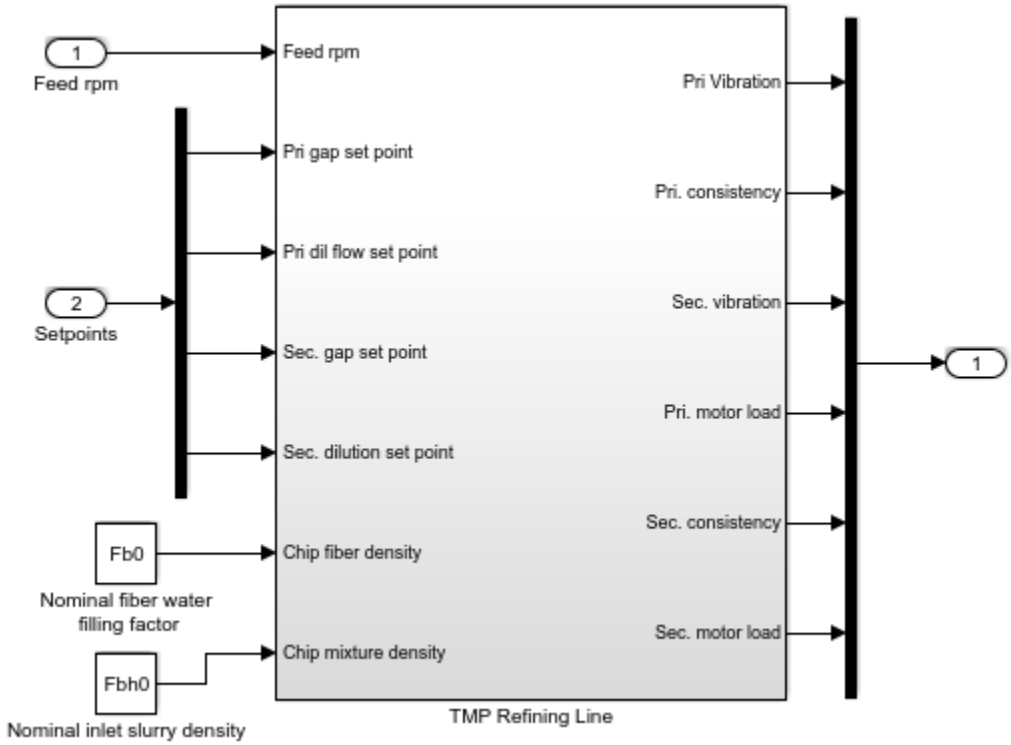
```
index2 =

     2     1
     2     2
     2     3
     2     4
```

Each row of `index2` is the index for one element of the Setpoints input vector.

Get Index of Specified Input Element of Operating Point Specification

Open Simulink model.

```
mdl = 'scdtmpSetpoints';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains specifications for the root-level input ports of the model.

Obtain the index of the element that corresponds to the second signal in the Setpoints input vector.

```
index1 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints',2)
```

```
index1 =
```

```
     2     2
```

You can also obtain the indices of multiple vector elements at the same time. For example, get the indices for the first and third elements of the Setpoints vector.

```
index2 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints',[1 3])
```

```
index2 =

     2     1
     2     3
```

Input Arguments

`op` — Operating point specification or operating point

`operspec` object | operating point object

Operating point specification or operating point for a Simulink model, specified as an `operspec` object or operating point object.

`block` — Block path

character vector | string

Block path that corresponds to an input specification in the `Inputs` property of `op`, specified as a character vector or string that contains the path of a root-level input of a Simulink model.

To see all the blocks that have input specifications, view the `Inputs` property of `op`.

```
op.Inputs
```

`element` — Input element index

positive integer | vector of positive integers

Input element index, specified as a positive integer less than or equal to the port width of the input specified by `block`, or as a vector of such integers. By default, if you do not specify `element`, `getInputIndex` returns the indices of all elements in the selected

input specification. For an example, see “Get Index of Specified Input Element of Operating Point Specification” on page 13-111.

Output Arguments

`index` — Input index

2-element row vector | 2-column array

Input index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single model input element.

The first column of `index` contains the index of the corresponding input specification in the `Inputs` property of `op`. The second column contains the element index within the input specification.

Using `index`, you can specify the input portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`findop` | `getOutputIndex` | `getStateIndex` | `operspec`

Topics

“Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50

Introduced in R2017a

getinputstruct

Input structure from operating point

Syntax

```
in_struct = getinputstruct(op_point)
```

Description

`in_struct = getinputstruct(op_point)` extracts a structure of input values, `in_struct`, from the operating point object, `op_point`. The structure, `in_struct`, uses the same format as Simulink software which allows you to set initial values for inputs in the model within the **Data Import/Export** pane of the Configuration Parameters dialog box.

Examples

Create an operating point object for the `scdplane` model:

```
open_system('scdplane')
op_scdplane = operpoint('scdplane');
```

Extract an input structure from the operating point object:

```
inputs_scdplane = getinputstruct(op_scdplane)

inputs_scdplane =

       time: 0
    signals: [1x1 struct]
```

To view the values of the inputs within this structure, use dot-notation to access the `values` field:

```
inputs_scdplane.signals.values
```

In this case, the value of the input is `0`.

See Also

`getstatestruct` | `getxu` | `operpoint`

Introduced before R2006a

getlinio

Obtain linear analysis points from Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
io = getlinio(mdl)
io = getlinio(blockpath)
```

Description

`io = getlinio(mdl)` returns the analysis points defined in the Simulink model `mdl`.

`io = getlinio(blockpath)` returns the analysis points defined for the specified Linear Analysis Plots block or Model Verification block in a Simulink model.

Examples

Obtain Analysis Points from Simulink Model

Open Simulink model.

```
mdl = 'scdpwm';
open_system(mdl)
```

This model contains the following linear analysis points:

- Input perturbation at the output of the Step block
- Output measurement at the output of the Plant Model block

Obtain the analysis points from the model.

```
io = getlinio(mdl)
```

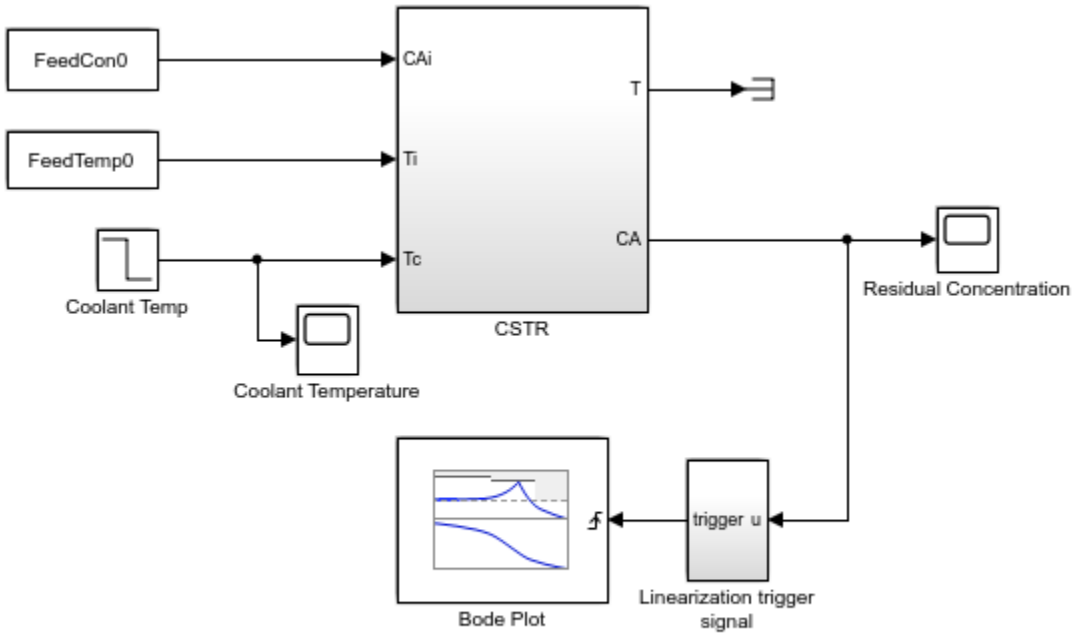
```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdpwm/Step
- Port: 1
2. Linearization output measurement located at the following signal:
- Block: scdpwm/Plant Model
- Port: 1
```

You can use these analysis points for subsequent linearizations of the model using the `linearize` command or an `slLinearizer` interface.

Obtain Analysis Points from Linear Analysis Plots Block

Open Simulink model.

```
open_system('scdcstr')
```



This model contains a Bode Plot block that is configured with the following linear analysis points:

- Input perturbation at the output of the Coolant Temp block
- Output measurement at the `CA` output of the CSTR block

Obtain the analysis points from the Bode Plot block.

```
io = getlinio('scdcstr/Bode Plot')
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdcstr/Coolant Temp
- Port: 1
2. Linearization output measurement located at the following signal:
```

```
- Block: scdcstr/CSTR
- Port: 2
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

If the model is not open or loaded into memory, `getlinio` loads the model into memory.

`blockpath` — Linear Analysis Plots block or Model Verification block

character vector | string

Linear Analysis Plots block or Model Verification block, specified as a character vector or string that contains its full block path. The model that contains the block must be in the current working folder or on the MATLAB path.

For more information on:

- Linear analysis plot blocks, see “Visualization During Simulation”.
- Model verification blocks, see “Model Verification”.

Output Arguments

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set, returned as a linearization I/O object or a vector of linearization I/O objects. Use `io` to specify linearization inputs, outputs, and loop openings when using the `linearize` command. For more information, see “Specify Portion of Model to Linearize” on page 2-13.

Each analysis point has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-13.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

See Also

`linearize` | `linio` | `setlinio`

Topics

“Specify Portion of Model to Linearize” on page 2-13

Introduced before R2006a

getlinplant

Compute open-loop plant model from Simulink diagram

Syntax

```
[sysp,sysc] = getlinplant(block,op)
[sysp,sysc] = getlinplant(block,op,options)
```

Description

`[sysp,sysc] = getlinplant(block,op)` Computes the open-loop plant seen by a Simulink block labeled `block` (where `block` specifies the full path to the block). The plant model, `sysp`, and linearized block, `sysc`, are linearized at the operating point `op`.

`[sysp,sysc] = getlinplant(block,op,options)` Computes the open-loop plant seen by a Simulink block labeled `block`, using the linearization options specified in `options`.

Examples

To compute the open-loop model seen by the Controller block in the Simulink model `magball`, first create an operating point object using the function `findop`. In this case, you find the operating point from simulation of the model.

```
magball
op=findop('magball',20);
```

Next, compute the open-loop model seen by the block `magball/Controller`, with the `getlinplant` function.

```
[sysp,sysc]=getlinplant('magball/Controller',op)
```

The output variable `sysp` gives the open-loop plant model as follows:

```
a =
            Current     dhdt    height
```

```
   Current     -100        0         0
   dhdt      -2.801        0     196.2
   height         0        1         0

b =
            Controller
   Current          50
   dhdt              0
   height            0

c =
          Current     dhdt   height
   Sum2         0        0       -1

d =
          Controller
   Sum2            0

Continuous-time model.
```

See Also

`findop` | `linearizeOptions` | `operpoint` | `operspec`

Introduced before R2006a

getOffsetsForLPV

Extract LPV offsets from linearization results

Syntax

```
offsets = getOffsetsForLPV(info)
```

Description

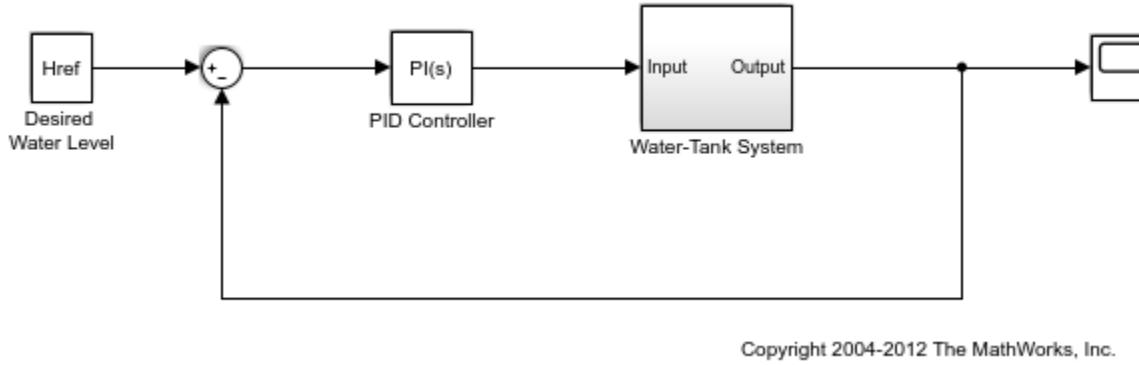
`offsets = getOffsetsForLPV(info)` extracts linearization offsets from `info` and converts them to the array format supported by the LPV System block.

Examples

Extract LPV Offsets from Linearization Results

Open the Simulink model.

```
model = 'watertank';
open_system(model)
```



Specify linearization I/Os.

```
io(1) = linio('watertank/Desired  Water Level',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

Vary plant parameters `A` and `b`, and create a 3-by-4 parameter grid.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),linspace(0.9*b,1.1*b,4));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a linearization option set, setting the `StoreOffsets` option to `true`.

```
opt = linearizeOptions('StoreOffsets',true);
```

Linearize the model using the specified parameter grid, and return the linearization offsets in the `info` structure.

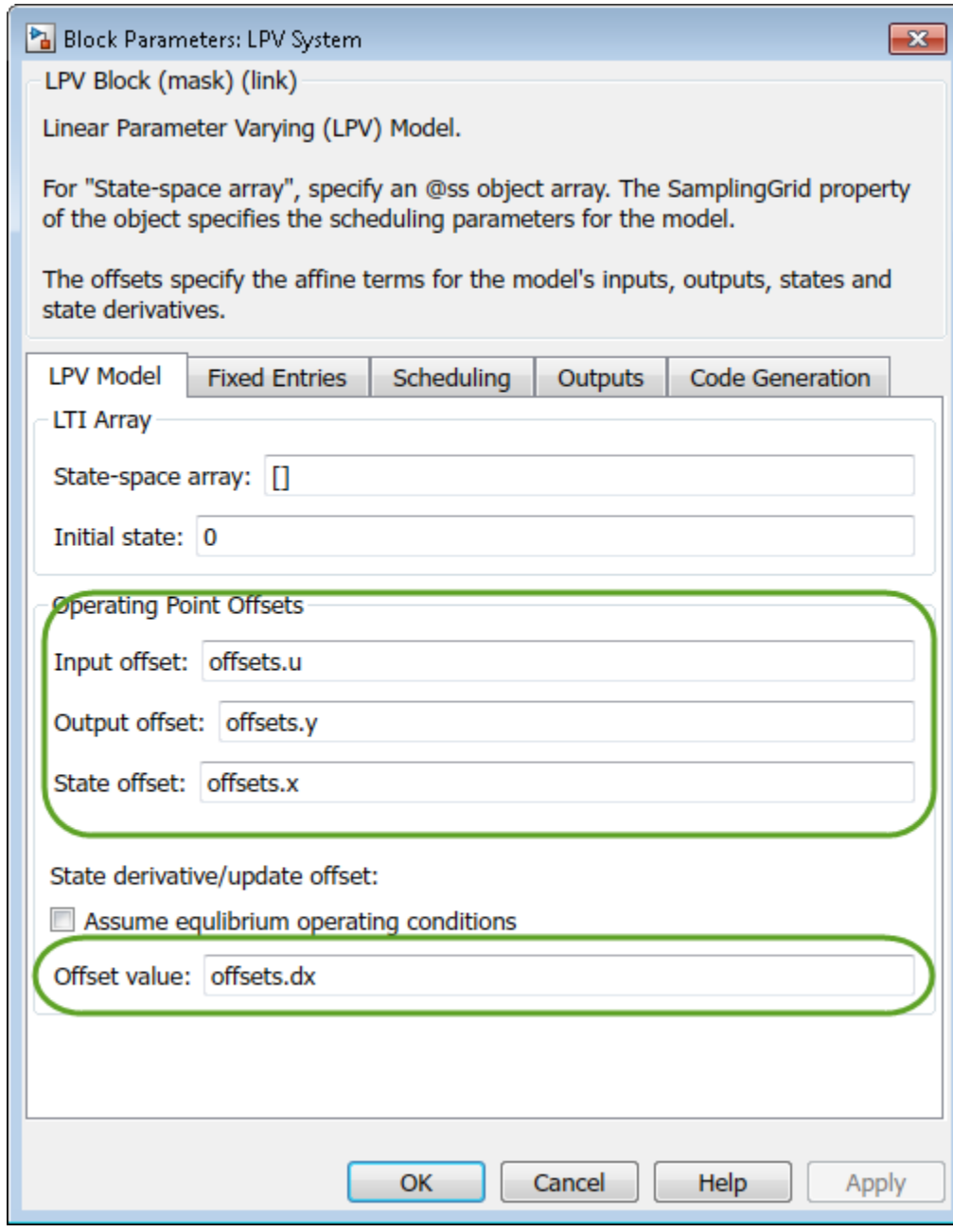
```
[sys,op,info] = linearize('watertank',io,params,opt);
```

Extract the linearization offsets.

```
offsets = getOffsetsForLPV(info)
```

```
offsets =

  struct with fields:

     x: [2x1x3x4 double]
     y: [1x1x3x4 double]
     u: [1x1x3x4 double]
    dx: [2x1x3x4 double]
```

To configure an LPV System block, use the fields from `offsets` directly.



Input Arguments

`info` — Linearization information

structure

Linearization information returned by exact linearization commands, specified as a structure. This structure has an `Offsets` field that contains an N_1-by-...-by-N_m array of structures, where N_1 to N_m are the dimensions of the operating point array or parameter grid used for linearization. Each structure in `info.Offsets` contains offset information that corresponds to a specific operating point.

You can store and obtain linearization offsets when you linearize your model using one of the following commands:

- `linearize`
- `getIOTransfer`
- `getLoopTransfer`
- `getSensitivity`
- `getCompSensitivity`

For example:

```
opt = linearizeOptions('StoreOffsets',true);
[sys,op,info] = linearize(mdl,io,params,opt);
```

You can then extract the offset information using `getOffsetsForLPV`.

```
offsets = getOffsetsForLPV(info);
```

Output Arguments

`offsets` — Linearization offsets

structure

Linearization offsets corresponding to the operating points at which the model was linearized, returned as a structure with the following fields:

Field	Description
`x`	State offsets used for linearization, returned as an n_x-by-1-by-N_1-by-...-by-N_m array, where n_x is the number of states in the linearized system.
`y`	Output offsets used for linearization, returned as an n_y-by-1-by-N_1-by-...-by-N_m array, where n_y is the number of outputs in the linearized system.

Field	Description
`u`	Input offsets used for linearization, returned as an n_u-by-1-by-N_1-by-. . .-by-N_m array, where n_u is the number of inputs in the linearized system.
`dx`	Derivative offsets for continuous time systems, or updated state values for discrete-time systems, returned as an n_x-by-1-by-N_1-by-. . .-by-N_m array.

For instance, suppose that your model has three inputs, two outputs, and four states. If you linearize your model using a 5-by-6 array of operating points, `offsets` contains arrays with the following dimensions:

- `offsets.x` — 4-by-1-by-5-by-6
- `offsets.y` — 2-by-1-by-5-by-6
- `offsets.u` — 3-by-1-by-5-by-6
- `offsets.dx` — 4-by-1-by-5-by-6

To configure an LPV System block, you can use the fields of `offsets` directly. For an example, see “Approximating Nonlinear Behavior Using an Array of LTI Systems” on page 3-91.

See Also

Blocks
LPV System

Functions
`getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize`

Topics
“Linear Parameter-Varying Models” (Control System Toolbox)
“Approximating Nonlinear Behavior Using an Array of LTI Systems” on page 3-91

Introduced in R2016b

getOutputIndex

Get index of an output element of an operating point specification

The `Outputs` property of an operating point specification is an array that contains trimming specifications for each model output. When defining a mapping function for customized trimming of Simulink models, `getOutputIndex` lets you obtain the index of an output specification based on the corresponding block path.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.

Syntax

```
index = getOutputIndex(op,block)
index = getOutputIndex(op,block,port)
index = getOutputIndex(op,block,port,element)
```

Description

`index = getOutputIndex(op,block)` returns the index of the output specification that corresponds to `block` in the `Outputs` property of operating point specification `op`.

`index = getOutputIndex(op,block,port)` returns the index of the output specification that corresponds to the trim output constraint added to the specified output `port` of the specified `block`.

Use this syntax when the`Outputs` property of `op` contains trim output constraints for more than one signal originating from the same block.

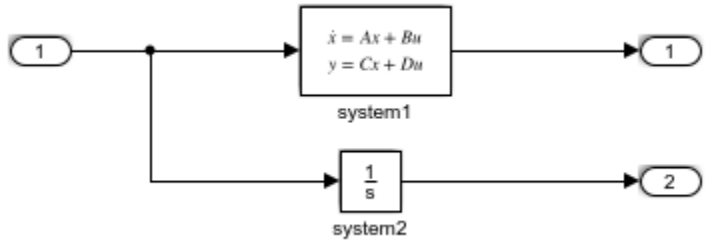
`index = getOutputIndex(op,block,port,element)` returns the index of the specified `element` within an output specification for an output with multiple elements.

Examples

Get Output Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification for model.

```
opspec = operspec(mdl);
```

`opspec` contains an array of output specifications for the model.

```
opspec.Outputs
```

```
(1.) scdindex1/Out1
     spec:  none
(2.) scdindex1/Out2
     spec:  none
```

Get the index of the output specification for Out2.

```
idx = getOutputIndex(opspec,'scdindex1/Out2')
```

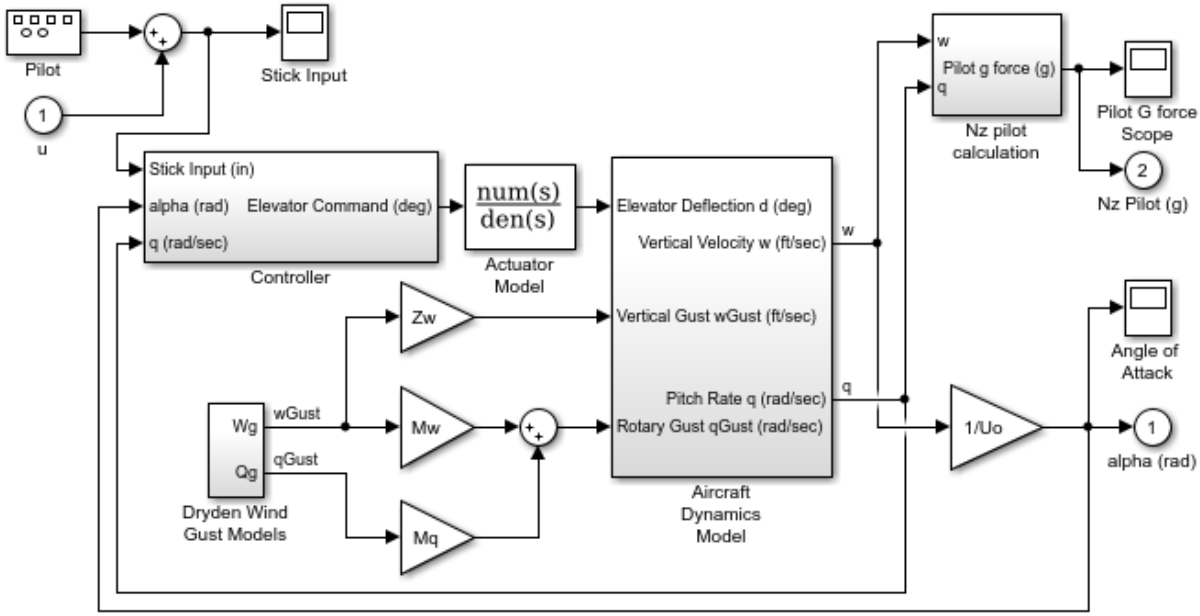
```
idx =

     2     1
```

The first column of `idx` contains the index of the output specification in `opspec.Outputs`. The second column contains the element index within the output specification. In this case, there is only one element in the output specification.

Get Index of Trim Output Specification Added To Signal

Open Simulink model.

```
mdl = 'scdplane';
open_system(mdl)
```



Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

In addition to root-level outputs of a model, the `opspec.Outputs` array contains specifications for trim constraints added to signals using the `addoutputspec` command.

Add an output specification to the signal originating from second output port of the Aircraft Dynamics Model block.

```
opspec = addoutputspec(opspec,'scdplane/Aircraft Dynamics Model',2);
```

View the output array of `opspec`.

```
opspec.Outputs
```

```
(1.) scdplane/alpha (rad)
     spec:  none
(2.) scdplane/Nz Pilot (g)
     spec:  none
(3.) scdplane/Aircraft Dynamics Model
     spec:  none
```

Get the index of the added output specification. When there is an output specification for only one of the output ports of a given block, you do not need to specify the port number to get the output index.

```
index1 = getOutputIndex(opspec,'scdplane/Aircraft Dynamics Model')
```

```
index1 =

     3     1
```

Add an output specification to the signal originating from the first output of the same block.

```
opspec = addoutputspec(opspec,'scdplane/Aircraft Dynamics Model',1);
```

View the output array of `opspec`.

```
opspec.Outputs
```

```
(1.) scdplane/alpha (rad)
     spec:  none
(2.) scdplane/Nz Pilot (g)
     spec:  none
(3.) scdplane/Aircraft Dynamics Model
```

```
     spec:  none
(4.) scdplane/Aircraft Dynamics Model
     spec:  none
```

There are now two output specifications that correspond to the same block, one for each output port. Obtain the index for the output specification that corresponds with the output port `1` of the Aircraft Dynamics Model block.

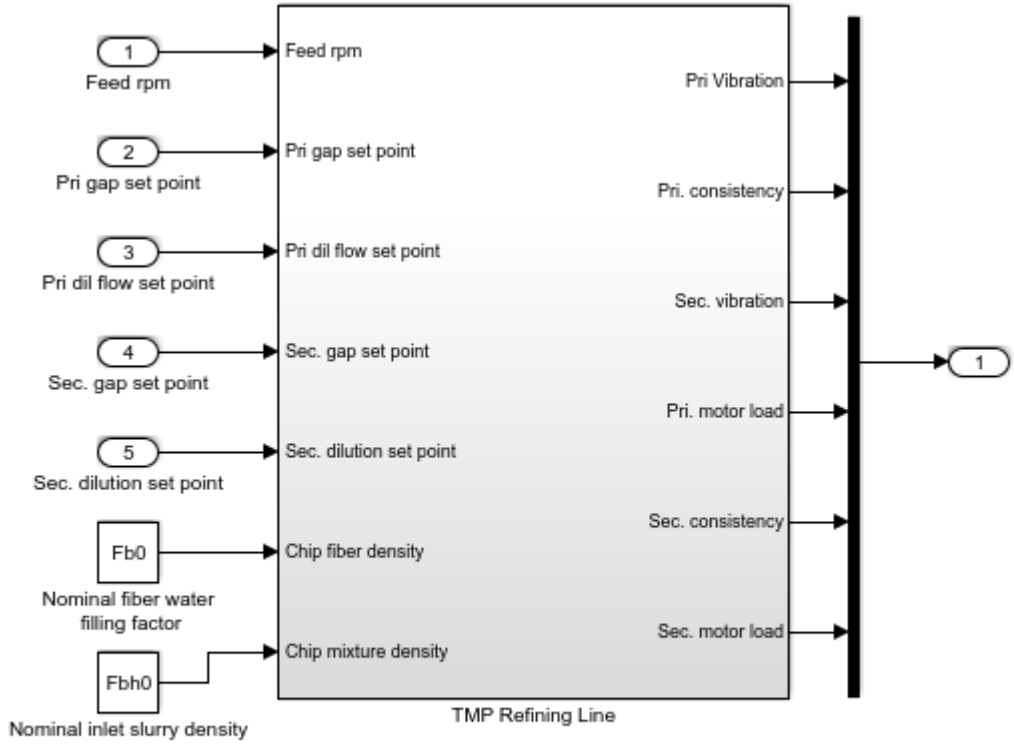
```
index2 = getOutputIndex(opspec,'scdplane/Aircraft Dynamics Model',1)
```

```
index2 =

     4     1
```

Get Output Indices for Output Specification with Multiple Elements

Open Simulink model.

```
mdl = 'scdtmp';
open_system(mdl)
```



Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains an output specification for the output port Out1, which is a vector signal.

```
opspec.Outputs
```

```
(1.) scdtmp/Out1
     spec:  none
     spec:  none
```

```
      spec:  none
      spec:  none
      spec:  none
      spec:  none
```

Obtain the indices of all the elements of Out1.

```
index1 = getOutputIndex(opspec,'scdtmp/Out1')
```

```
index1 =

     1     1
     1     2
     1     3
     1     4
     1     5
     1     6
```

Each row of `index1` contains the index for one element of the vector signal in Out1. The first column is the index of the output specification object for the Out1 port in the `opsepc.Outputs`. The second column is the element index within the output specification.

You can also obtain the index for individual elements of an output specification, or a subset of elements. Get the index of element number `4` of Out1.

```
index2 = getOutputIndex(opspec,'scdtmp/Out1',[],4)
```

```
index2 =

     1     4
```

Get the indices of elements `2` and `3` of Out1.

```
index3 = getOutputIndex(opspec,'scdtmp/Out1',[],[2 3])
```

```
index3 =

     1     2
```

```
     1     3
```

Input Arguments

`op` — Operating point specification

`operspec` object

Operating point specification for a Simulink model, specified as an `operspec` object.

`block` — Block path

character vector | string

Block path that corresponds to an output specification in the `Outputs` property of `op`, specified as a character vector or string that contains the path of one of the following:

- Root-level output of the model.
- Source block for a signal in the model to which an output specification has been added. For more information on adding output specifications to a model, see `addoutputspec`.

To see all the blocks that have output specifications, view the `Outputs` property of `op`.

```
op.Outputs
```

`port` — Output port

integer in the range `[1,N]`

Output port, specified as an integer in the range `[1,N]`, where `N` is the number of output ports on the specified `block`. If `block` is a root-level output port, then `N` is `1`.

If you do not specify `port`, and there is one entry in the output array of `op` that corresponds to the specified `block`, then the default value of `port` is the port number of that entry. If there are multiple entries in the output array that correspond to the specified `block`, then the default value of `port` is the port number of the first entry. For an example, see “Get Index of Trim Output Specification Added To Signal” on page 13-132.

To view the port number of the `i`th entry in the output array of `op`, type:

```
op.Outputs(i).PortNumber
```

`element` — Output element index

`[1,M]` (default) | positive integer | vector of positive integers

Output element index, specified as a positive integer less than or equal to the port width of the output of the specified `block`, or a vector of such integers. By default, if you do not specify `element`, `getOutputIndex` returns the indices of all elements in the selected output specification. For an example, see "Get Output Indices for Output Specification with Multiple Elements" on page 13-134.

Output Arguments

`index` — Output index

2-element row vector | 2-column array

Output index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single output element.

The first column of `index` contains the index of the corresponding output specification in the `Outputs` property of `op`. The second column contains the element index within the output specification.

Using `index`, you can specify the output portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`findop` | `getInputIndex` | `getStateIndex` | `operspec`

Topics

"Compute Operating Points Using Custom Constraints and Objective Functions" on page 1-50

Introduced in R2017a

getSimulationTime

Final time of simulation for frequency response estimation

Syntax

```
tfinal = getSimulationTime(input)
```

Description

`tfinal = getSimulationTime(input)` returns the final time of the Simulink simulation performed during frequency response estimation using the input signal `input`. Altering `input` to reduce the final simulation time can help reduce the time it takes to perform frequency response estimation.

Input Arguments

`input`

Input signal for frequency response estimation with the `frestimate` command.

The input signal `input` must be either:

- A sinestream input signal, created in the Linear Analysis Tool or created with `frest.Sinestream`
- A chirp input signal, created in the Linear Analysis Tool or created with `frest.Chirp`
- A random input signal, created in the Linear Analysis Tool or created with `frest.Random`

Output Arguments

`tfinal`

Final time of simulation performed during frequency response estimation using the input signal `input`.

For example, the command `sysest = frestimate(mdl,io,input)` performs frequency response estimation on the Simulink model specified by `mdl` with the linearization I/O set `io`. The estimation uses the input signal `input`. The command `tfinal = getSimulationTime(input)` returns the simulation time at the end of the simulation performed by `frestimate`.

Examples

Retrieve Simulation Time for Frequency Response Estimation

Create a sinestream input signal.

```
input = frest.Sinestream('Amplitude',1e-3,...
                         'Frequency',logspace(1,3,50),...
                         'SamplesPerPeriod',40,'FreqUnits','Hz');
```

The sinestream signal `input` includes 50 frequencies spaced logarithmically between 10 Hz and 1000 Hz. Each frequency is sampled 40 times per period.

Calculate the final simulation time of an estimation using that signal.

```
tfinal = getSimulationTime(input)
```

```
tfinal =

    4.4186
```

`tfinal` indicates that frequency response estimation of any model with this input signal would simulate the model for 4.4186 s.

- "Create Sinestream Input Signals" on page 5-13
- "Create Chirp Input Signals" on page 5-19

See Also

`frest.Chirp` | `frest.Random` | `frest.Sinestream` | `frestimate`

Topics

"Create Sinestream Input Signals" on page 5-13
"Create Chirp Input Signals" on page 5-19
"Ways to Speed up Frequency Response Estimation" on page 5-78

Introduced in R2012a

getStateIndex

Get index of a state element of an operating point specification or operating point

The `States` property of an operating point specification is an array that contains trimming specifications for each model state. When defining a mapping function for customized trimming of Simulink models, `getStateIndex` lets you obtain the index of a state specification based on the corresponding block path or state name.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.

Syntax

```
index = getStateIndex(op,name)
index = getStateIndex(op,name,element)
```

Description

`index = getStateIndex(op,name)` returns the index of the state specification that corresponds to `name` in the `States` property of operating point specification `op`.

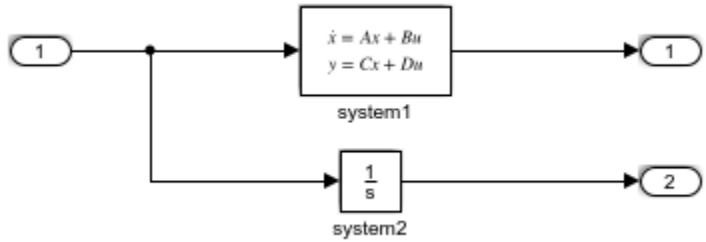
`index = getStateIndex(op,name,element)` returns the index of the specified `element` within a state specification for a block with multiple states.

Examples

Get State Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification for model.

```
opspec = operspec(mdl);
```

`opspec` contains an array of state specifications for the model.

```
opspec.States
```

```
(1.) scdindex1/system1
     spec:  dx = 0,  initial guess: 0
     spec:  dx = 0,  initial guess: 0
     spec:  dx = 0,  initial guess: 0
(2.) scdindex1/system2
     spec:  dx = 0,  initial guess: 0
```

Get the index of the state specification that corresponds to the system2 block.

```
index2 = getStateIndex(opspec,'scdindex1/system2')
```

```
index2 =

     2     1
```

`index2(1)` is the index of the state specification object for system2 in `opspec.States`. Since this block has a single state, `index2` has a single row and `index2(2)` is `1`.

If a block has multiple states, you can obtain the indices of all the states in the corresponding state specification.

```
index1 = getStateIndex(opspec,'scdindex1/system1')
```

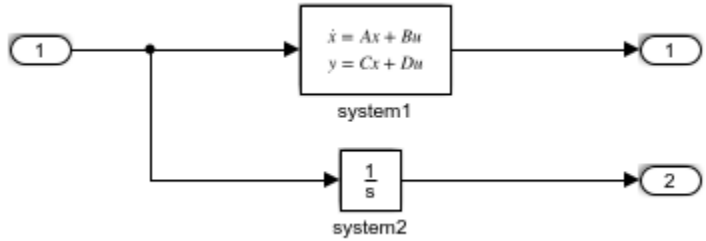
```
index1 =

     1     1
     1     2
     1     3
```

Each row of `index1` contains the index of one state in the system2 block. For each row, the first column contains the index of the state specification in `opspec.States`. The second column contains the index of each state element within the specification.

Get Index of Specified State Element of Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

If a block has multiple states, you can obtain the index of a specific state within the corresponding state specification by specifying the element index. For example, get the index for the second state in the specification for the system1 block.

```
index1 = getStateIndex(opspec,'scdindex1/system1',2)
```

```
index1 =

     1     2
```

You can also obtain the indices of a subset of the block states by specfying the element index as a vector. For example, get the indices for the first and third states in the specification for the system1 block.

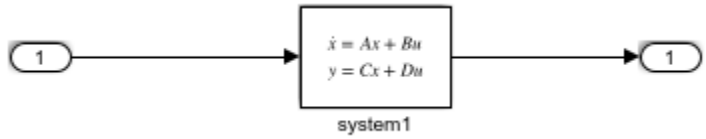
```
index2 = getStateIndex(opspec,'scdindex1/system1',[1 3])
```

```
index2 =

     1     1
     1     3
```

Get Index of Named State from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex2';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



The system1 block is a state-space system with three named states: `position`, `velocity`, and `acceleration`.

Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

The `States` property of the operating point specification object contains one entry for each named state in system1.

```
opspec.States
```

```
(1.) position
     spec:  dx = 0,  initial guess: 0
(2.) velocity
     spec:  dx = 0,  initial guess: 0
(3.) acceleration
     spec:  dx = 0,  initial guess: 0
```

To obtain the index of a state specification that corresponds to a named state within a block, specify the state name.

```
index1 = getStateIndex(opspec,'velocity')
```

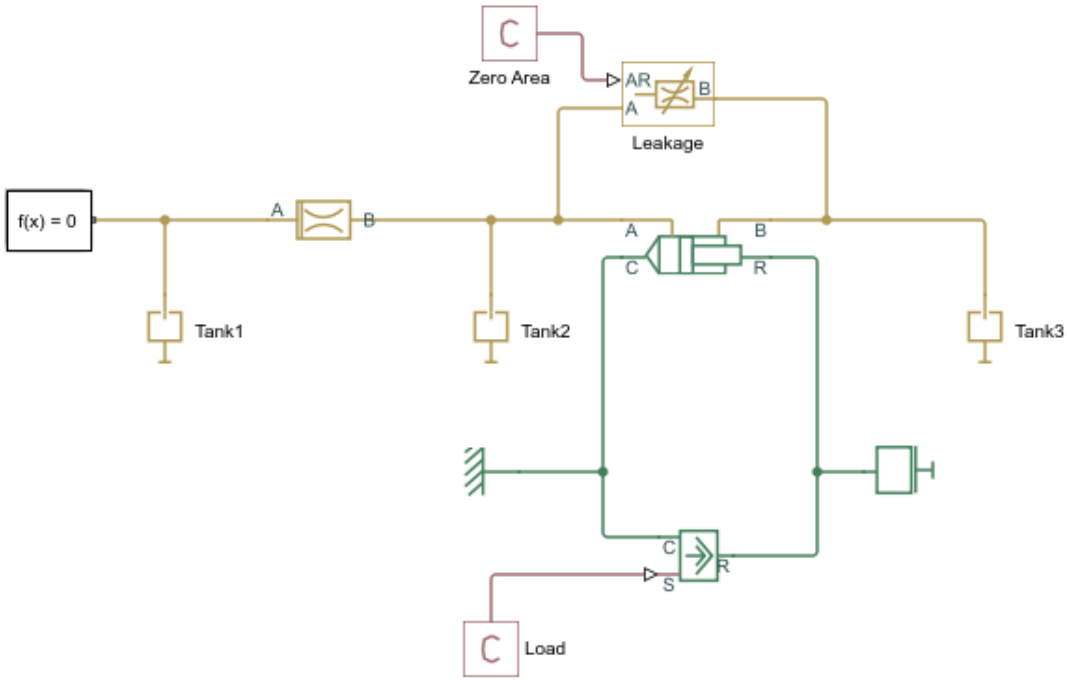
```
index1 =

     2     1
```

The first column of `index1` contains the index of the corresponding state specification in the `opspec.States` property. The second column is `1` for a named state.

Get Index of Simscape State from Operating Point Specification

Open model.

```
mdl = 'scdTanks_simscape';
mdlpath = fullfile(matlabroot,'examples','slcontrol',mdl);
open_system(mdlpath)
```



Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

The `States` property of the operating point specification object contains one state specification for each Simscape state in the model.

To obtain the index of a specification that corresponds to a Simscape state, specify the state name. For example, get the index of the pressure state of Tank3.

```
idx = getStateIndex(opspec,'scdTanks_simscape.Tank3.pressure')
```

```
idx =

    21     1
```

The first column of `idx` contains the index of the corresponding state specification in `opspec.States`. The second column is `1` for a Simscape state.

View the specification in `opspec.States` for this state.

```
opspec.States(idx(1))
```

```
(1.) scdTanks_simscape.Tank3.pressure
     spec:  dx = 0,  initial guess: 0
```

Input Arguments

`op` — Operating point specification or operating point

`operspec` object | operating point object

Operating point specification or operating point for a Simulink model, specified as an `operspec` object or operating point object.

`name` — Block path or state name

character vector | string

Block path or state name that corresponds to a state specification in the `States` property of `op`, specified as a character vector or string that contains one of the following:

- Block path of a block in the Simulink model that contains unnamed states.
- Name of a named state in a Simulink or Simscape block.

To see all the states that have state specifications, view the `States` property of `op`.

```
op.States
```

`element` — State element index

positive integer | vector of positive integers

State element index, specified as a positive integer less than or equal to the number of state elements in the block or state specified by `name`, or a vector of such integers. By default, if you do not specify `element`, `getStateIndex` returns the indices of all elements in the selected state specification. For an example, see “Get Index of Specified State Element of Operating Point Specification” on page 13-144.

Output Arguments

`index` — State index

2-element row vector | 2-column array

State index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single model state.

The first column of `index` contains the index of the corresponding state specification in the `States` property of `op`. The second column contains the element index within the state specification.

Using `index`, you can specify the state portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`findop` | `getInputIndex` | `getOutputIndex` | `operspec`

Topics

“Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50

Introduced in R2017a

getstatestruct

State structure from operating point

Syntax

```
x_struct = getstatestruct(op_point)
```

Description

`x_struct = getstatestruct(op_point)` extracts a structure of state values, `x_struct`, from the operating point object, `op_point`. The structure, `x_struct`, uses the same format as Simulink software which allows you to set initial values for states in the model within the **Data Import/Export** pane of the Configuration Parameters dialog box.

Examples

Create an operating point object for the `magball` model:

```
op_magball=operpoint('magball');
```

Extract a state structure from the operating point object:

```
states_magball=getstatestruct(op_magball)
```

This extraction returns

```
states_magball =

       time: 0
    signals: [1x5 struct]
```

To view the values of the states within this structure, use dot-notation to access the `values` field:

```
states_magball.signals.values
```

This dot-notation returns

```
ans =

     0


ans =

   14.0071


ans =

    7.0036


ans =

     0


ans =

    0.0500
```

See Also

`getinputstruct` | `getxu` | `operpoint`

Introduced before R2006a

getxu

States and inputs from operating points

Syntax

```
x = getxu(op_point)
[x,u] = getxu(op_point)
[x,u,xstruct] = getxu(op_point)
```

Description

`x = getxu(op_point)` extracts a vector of state values, `x`, from the operating point object, `op_point`. The ordering of states in `x` is the same as that used by Simulink software.

`[x,u] = getxu(op_point)` extracts a vector of state values, `x`, and a vector of input values, `u`, from the operating point object, `op_point`. States in `x` and inputs in `u` are ordered in the same way as for Simulink.

`[x,u,xstruct] = getxu(op_point)` extracts a vector of state values, `x`, a vector of input values, `u`, and a structure of state values, `xstruct`, from the operating point object, `op_point`. The structure of state values, `xstruct`, has the same format as that returned from a Simulink simulation. States in `x` and `xstruct` and inputs in `u` are ordered in the same way as for Simulink.

Examples

Create an operating point object for the `magball` model by typing:

```
op=operpoint('magball');
```

To view the states within this operating point, type:

```
op.States
```

which returns

```
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
```

To extract a vector of state values, with the states in an ordering that is compatible with Simulink, along with inputs and a state structure, type:

```
[x,u,xstruct]=getxu(op)
```

This syntax returns:

```
x =

    0.0500
         0
   14.0071
    7.0036
         0


u =

     []


xstruct =

       time: 0
    signals: [1x5 struct]
```

View `xstruct` in more detail by typing:

```
xstruct.signals
```

This syntax displays:

```
ans =

1x5 struct array with fields:
    values
    dimensions
    label
    blockName
    stateName
    inReferencedModel
    sampleTime
```

View each component of the structure individually. For example:

```
xstruct.signals(1).values

ans =

     0
```

or

```
xstruct.signals(2).values

ans =

    7.0036
```

You can import these vectors and structures into Simulink as initial conditions or input vectors or use them with `setxu`, to set state and input values in another operating point.

See Also

`operpoint` | `operspec`

Introduced before R2006a

highlight

Package: linearize.advisor

Highlight linearization path in Simulink model

Syntax

```
highlight(advisor)
```

Description

`highlight(advisor)` highlights the linearization path for the model linearization associated with a `LinearizationAdvisor` object. The software identifies blocks that are on or off the linearization path. Also, for blocks that are on the linearization path, the software indicates whether they contribute to the linearization result.

Examples

Highlight Linearization Path

Load Simulink model.

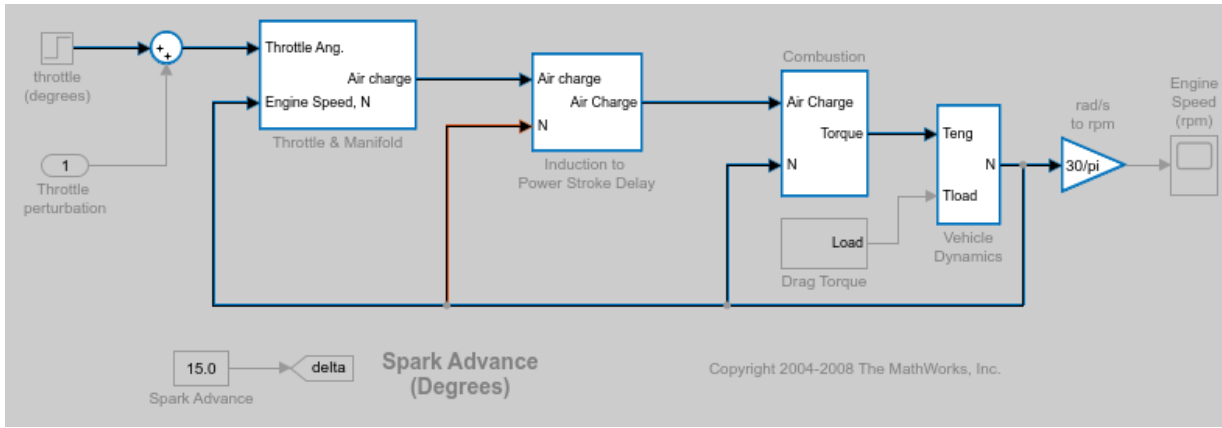
```
mdl = 'scdspeed';
load_system(mdl)
```

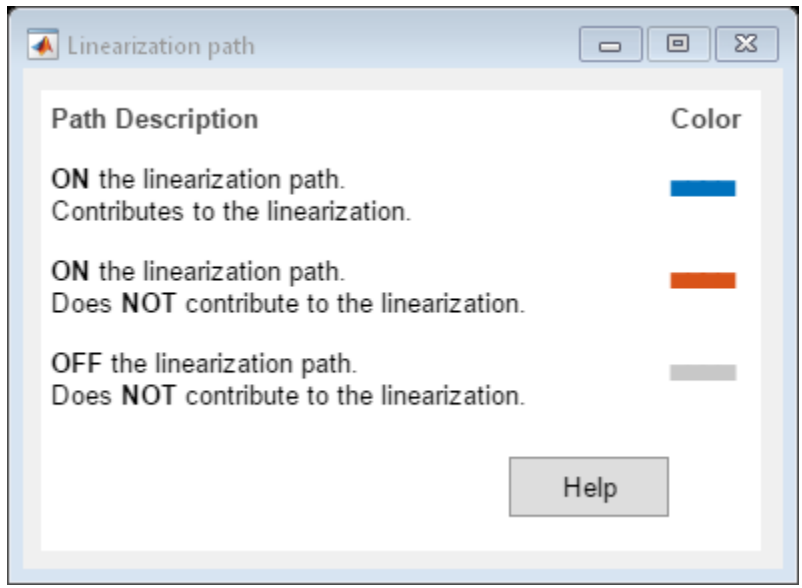
Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Highlight the linearization path.

```
highlight(advisor)
```





The Simulink model is highlighted as follows:

- Blue blocks are on the linearization path and contribute to the model linearization.
- Red blocks are on the linearization path and do not contribute to the model linearization.

- Gray blocks are not on the linearization path.

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object.

See Also

Using Objects

`LinearizationAdvisor`

Functions

`advise` | `find` | `getBlockInfo`

Topics

"Troubleshoot Linearization Results at Command Line" on page 4-40

Introduced in R2017b

initopspec

Initialize operating point specification values

Syntax

```
opnew=initopspec(opspec,oppoint)
opnew=initopspec(opspec,x,u)
opnew=initopspec(opspec,xstruct,u)
```

Description

`opnew=initopspec(opspec,oppoint)` initializes the operating point specification object, `opspec`, with the values contained in the operating point object, `oppoint`. The function returns a new operating point specification object, `opnew`. Create `opspec` with the function `operspec`. Create `oppoint` with the function `operpoint` or `findop`.

`opnew=initopspec(opspec,x,u)` initializes the operating point specification object, `opspec`, with the values contained in the state vector, `x`, and the input vector, `u`. The function returns a new operating point specification object, `opnew`. Create `opspec` with the function `operspec`. You can use the function `getxu` to create `x` and `u` with the correct ordering.

`opnew=initopspec(opspec,xstruct,u)` initializes the operating point specification object, `opspec`, with the values contained in the state structure, `xstruct`, and the input vector, `u`. The function returns a new operating point specification object, `opnew`. Create `opspec` with the function `operspec`. You can use the function `getstatestruct` or `getxu` to create `xstruct` and the function `getxu` to create `u` with the correct ordering. Alternatively, you can save `xstruct` to the MATLAB workspace after a simulation of the model. See the Simulink documentation for more information on these structures.

Examples

Create an operating point using `findop` by simulating the `magball` model and extracting the operating point after `20` time units.

```
oppoint=findop('magball',20)
```

This syntax returns the following operating point:

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=20)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 2.33e-007
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 3.6e-008
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

Use these operating point values as initial values in an operating point specification object.

```
opspec=operspec('magball');
newopspec=initopspec(opspec,oppoint)
```

The new operating point specification object is displayed.

```
 Operating Specification for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      spec:  dx = 0,  initial guess:     2.33e-007
(2.) magball/Controller/PID Controller/Integrator
      spec:  dx = 0,  initial guess:            14
(3.) magball/Magnetic Ball Plant/Current
      spec:  dx = 0,  initial guess:             7
(4.) magball/Magnetic Ball Plant/dhdt
      spec:  dx = 0,  initial guess:      3.6e-008
(5.) magball/Magnetic Ball Plant/height
```

```
      spec:  dx = 0,  initial guess:          0.05

Inputs: None
----------

Outputs: None
----------
```

You can now use this object to find operating points by optimization.

Alternatives

As an alternative to the `initopspec` function, initialize operating point specification values in the Linear Analysis Tool. See “Import and Export Specifications For Operating Point Search” on page 1-48.

See Also

`findop` | `getstatestruct` | `getxu` | `operpoint` | `operspec`

Introduced before R2006a

linearize

Linear approximation of Simulink model or subsystem

Syntax

```
linsys = linearize(sys,io)
linsys = linearize(sys,io,op)
linsys = linearize(sys,io,tsnapshot)

linsys = linearize(sys,blockpath)
linsys = linearize(sys,blockpath,op)
linsys = linearize(sys,blockpath,tsnapshot)

linsys = linearize(sys,blocksub,io)
linsys = linearize(sys,blocksub,io,op)
linsys = linearize(sys,blocksub,io,tsnapshot)

linsys = linearize(___,param)

linsys = linearize(___,'StateOrder',stateorder)

linsys = linearize(___,options)

[linsys,linop] = linearize(___)

[linsys,linop,info] = linearize(___)
```

Description

`linsys = linearize(sys,io)` returns a linear approximation of the nonlinear Simulink model, `sys`, at the model operating point using the analysis points specified in `io`. If you omit `io`, then `linearize` uses the root level inports and outports of the model as analysis points.

`linsys = linearize(sys,io,op)` linearizes the model at operating point, `op`. To compute operating points from specifications or at simulation snapshot times, use `findop`.

`linsys = linearize(sys,io,tsnapshot)` simulates the system and linearizes the model at the specified snapshot times, `tsnapshot`.

`linsys = linearize(sys,blockpath)` linearizes the block or subsystem in `sys` specified by `blockpath` at the model operating point. The software isolates the block from the rest of the model before linearization.

`linsys = linearize(sys,blockpath,op)` linearizes the specified block at operating point `op`.

`linsys = linearize(sys,blockpath,tsnapshot)` simulates the system and linearizes the specified block at the specified snapshot times, `tsnapshot`.

`linsys = linearize(sys,blocksub,io)` linearizes `sys` using the substitute block or subsystem linearizations specified in `blocksub`. The linearization uses the model operating point and analysis point set `io`.

`linsys = linearize(sys,blocksub,io,op)` linearizes `sys` at operating point `op` using the specified block linearizations.

`linsys = linearize(sys,blocksub,io,tsnapshot)` simulates the system and linearizes the model at the specified snapshot times, `tsnapshot`, using the specified block linearizations.

`linsys = linearize(___,param)` linearizes the model using the parameter value variations specified in `param`. You can vary any model parameter with a value given by a variable in the model workspace, the MATLAB workspace, or a data dictionary.

`linsys = linearize(___,'StateOrder',stateorder)` specifies the order of the states in the linearized model.

`linsys = linearize(___,options)` linearizes the model using additional linearization options.

`[linsys,linop] = linearize(___)` returns the operating point at which the model was linearized. Obtain linearization operating points when linearizing at simulation snapshots, or when varying parameters during linearization.

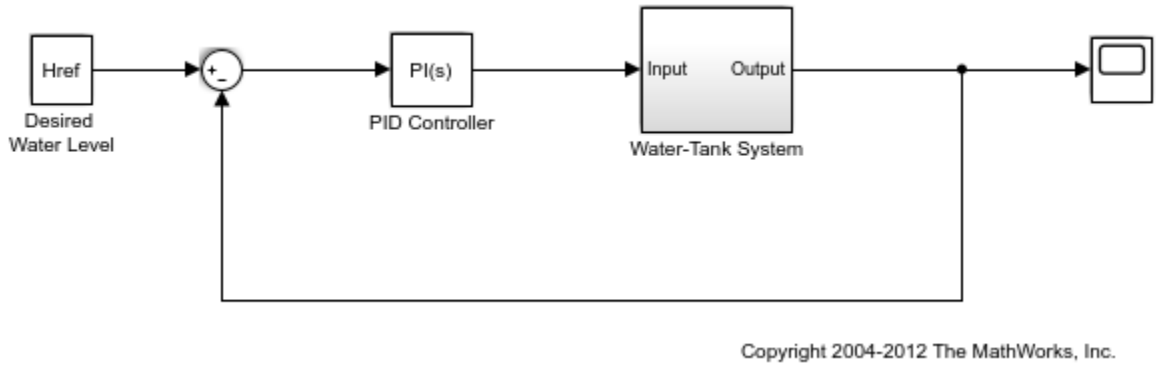
`[linsys,linop,info] = linearize(___)` returns additional linearization information. To select the linearization information to return in `info`, specify the corresponding option in `options`.

Examples

Linearize Model Using Specified I/O Set

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Specify a linearization input at the output of the PID Controller block, which is the input signal for the Water-Tank System block.

```
io(1) = linio('watertank/PID Controller',1,'input');
```

Specify a linearization output point at the output of the Water-Tank System block. Specifying the output point as open-loop removes the effects of the feedback signal on the linearization without changing the model operating point.

```
io(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

Linearize the model using the specified I/O set.

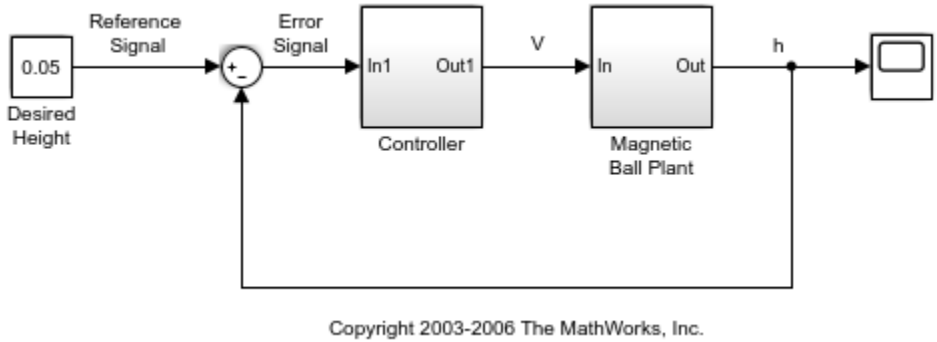
```
linsys = linearize(sys,io);
```

`linsys` is the linear approximation of the plant at the model operating point.

Linearize Model at Specified Operating Point

Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```



Find a steady-state operating point at which the ball height is `0.05`. Create a default operating point specification, and set the height state to a known value.

```
opspec = operspec(sys);
opspec.States(5).Known = 1;
opspec.States(5).x = 0.05;
```

Trim the model to find the operating point.

```
options = findopOptions('DisplayReport','off');
op = findop(sys,opspec,options);
```

Specify linearization input and output signals to compute the closed-loop transfer function.

```
io(1) = linio('magball/Desired Height',1,'input');
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

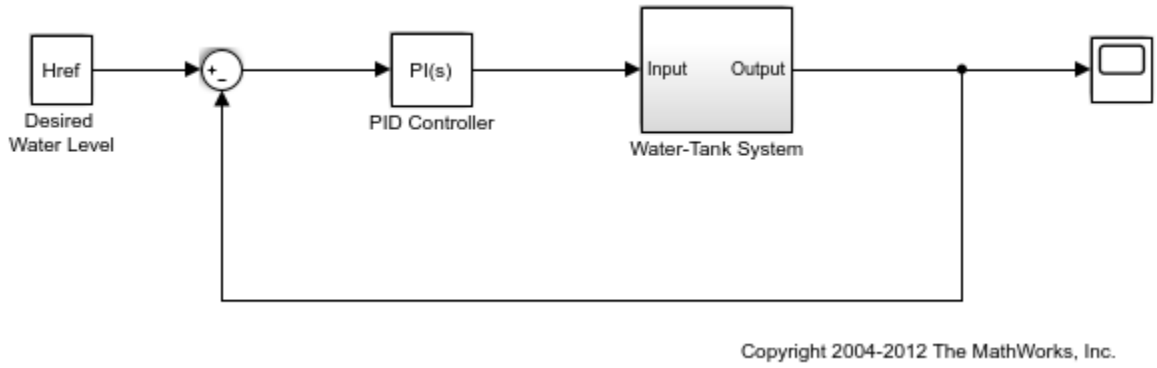
Linearize the model at the specified operating point using the specified I/O set.

```
linsys = linearize(sys,io,op);
```

Linearize Block or Subsystem at Model Operating Point

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Specify the full block path for the block you want to linearize.

```
blockpath = 'watertank/Water-Tank System';
```

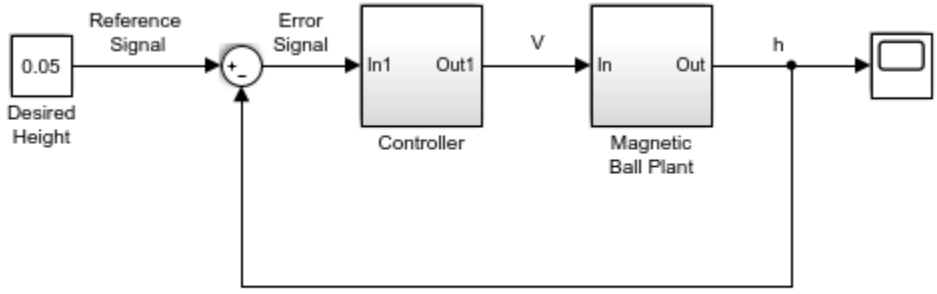
Linearize the specified block at the model operating point.

```
linsys = linearize(sys,blockpath);
```

Linearize Block or Subsystem at Trimmed Operating Point

Open Simulink model.

```
sys = 'magball';
open_system(sys)
```



Find a steady-state operating point at which the ball height is `0.05`. Create a default operating point specification, and set the height state to a known value.

```
opspec = operspec(sys);
opspec.States(5).Known = 1;
opspec.States(5).x = 0.05;

options = findopOptions('DisplayReport','off');
op = findop(sys,opspec,options);
```

Specify the block path for the block you want to linearize.

```
blockpath = 'magball/Magnetic Ball Plant';
```

Linearize the specified block and the specified operating point.

```
linsys = linearize(sys,blockpath,op);
```

Specify Substitute Block Linearization and Linearize Model

Open the Simulink model.

```
sys = 'scdpwm';
open_system(sys)
```

Extract linearization input and output from the model.

```
io = getlinio(sys);
```

Linearize the model at the model operating point.

```
linsys = linearize(sys,io)
```

```
linsys =

  D =
                 Step
   Plant Model      0

Static gain.
```

The discontinuities in the Voltage to PWM block cause the model to linearize to zero. To treat this block as a unit gain during linearization, specify a substitute linearization for this block.

```
blocksub.Name = 'scdpwm/Voltage to PWM';
blocksub.Value = 1;
```

Linearize the model using the specified block substitution.

```
linsys = linearize(sys,blocksub,io)
```

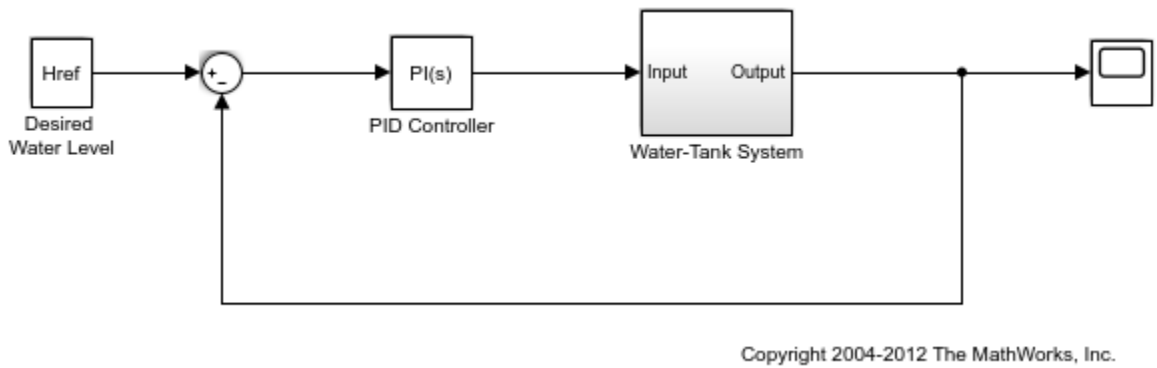
```
linsys =

  A =
                  State Space(  State Space(
   State Space(        0.9999       -0.0001
   State Space(        0.0001             1
```

```
  B =
                    Step
   State Space(   0.0001
   State Space(    5e-09

  C =
                State Space(   State Space(
   Plant Model              0              1

  D =
                Step
   Plant Model     0

Sample time: 0.0001 seconds
Discrete-time state-space model.
```

Linearize Model at Simulation Snapshot Time

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



To compute the closed-loop transfer function, first specify the linearization input and output signals.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

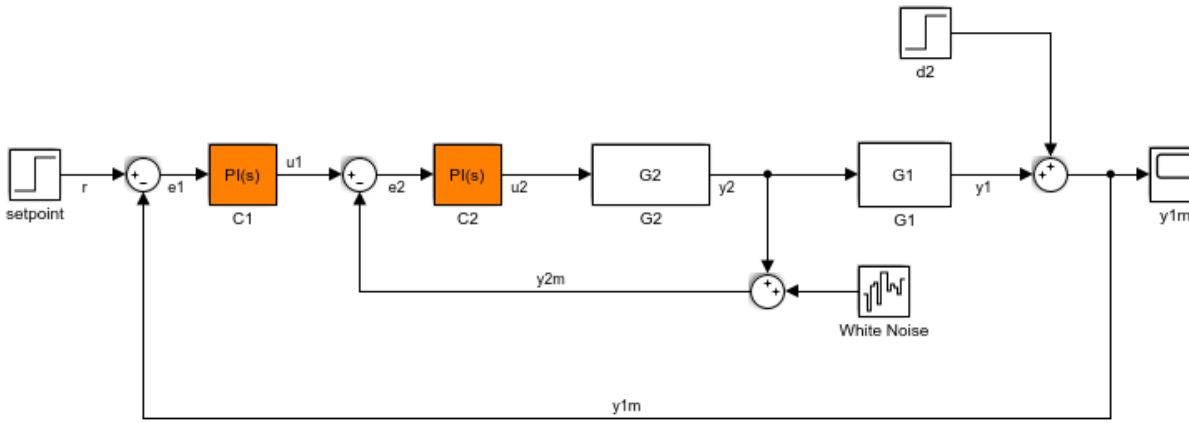
Simulate `sys` for `10` seconds and linearize the model.

```
linsys = linearize(sys,io,10);
```

Batch Linearize Model for Parameter Variations

Open the Simulink model.

```
sys = 'scdcascade';
open_system(sys)
```



Specify parameter variations for the outer-loop controller gains, `Kp1` and `Ki1`. Create parameter grids for each gain value.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid,Ki1_grid] = ndgrid(Kp1_range,Ki1_range);
```

Create a parameter value structure with fields `Name` and `Value`.

```
params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;
```

`params` is a 6-by-4 parameter value grid, where each grid point corresponds to a unique combination of `Kp1` and `Ki1` values.

Define linearization input and output points for computing the closed-loop response of the system.

```
io(1) = linio('scdcascade/setpoint',1,'input');
io(2) = linio('scdcascade/Sum',1,'output');
```

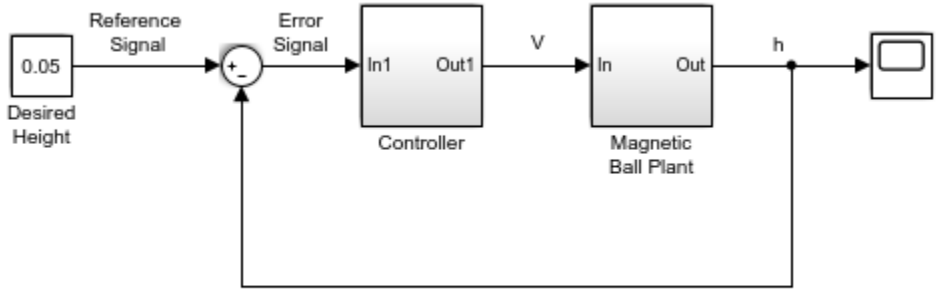
Linearize the model at the model operating point using the specified parameter values.

```
linsys = linearize(sys,io,params);
```

Specify State Order in Linearized Model

Open the Simulink model.

```
sys = 'magball';
open_system(sys)
```



Linearize the plant at the model operating point.

```
blockpath = 'magball/Magnetic Ball Plant';
linsys = linearize(sys,blockpath);
```

View the default state order for the linearized plant.

```
linsys.StateName
```

```
ans =

  3x1 cell array

    {'height' }
    {'Current'}
    {'dhdt'   }
```

Linearize the plant and reorder the states in the linearized model. Set the rate of change of the height as the second state.

```
stateorder = {'magball/Magnetic Ball Plant/height';...
              'magball/Magnetic Ball Plant/dhdt';...
              'magball/Magnetic Ball Plant/Current'};
linsys = linearize(sys,blockpath,'StateOrder',stateorder);
```

View the new state order.

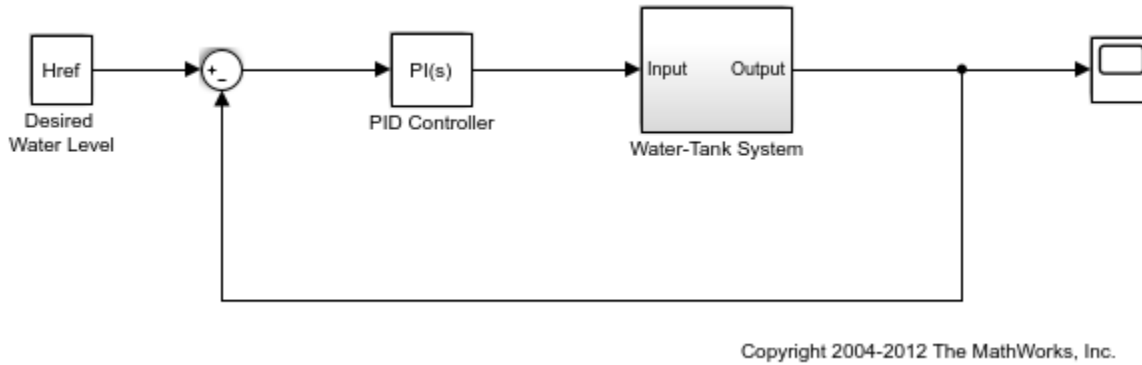
```
linsys.StateName
```

```
ans =

  3x1 cell array

    {'height' }
    {'dhdt'   }
    {'Current'}
```

Specify Sample Time of Linearized Model

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Specify the block path of the plant.

```
blockpath = 'watertank/Water-Tank System';
```

Create a linearization option set, and specify the sample time for the linearized model.

```
options = linearizeOptions('SampleTime',0.1);
```

Linearize the plant using the specified options.

```
linsys = linearize(sys,blockpath,options)
```

```
linsys =

  A =
           H
   H   0.995

  B =
      Water-Tank S
   H       0.02494

  C =
                  H
   Water-Tank S   1

  D =
                  Water-Tank S
   Water-Tank S              0
```

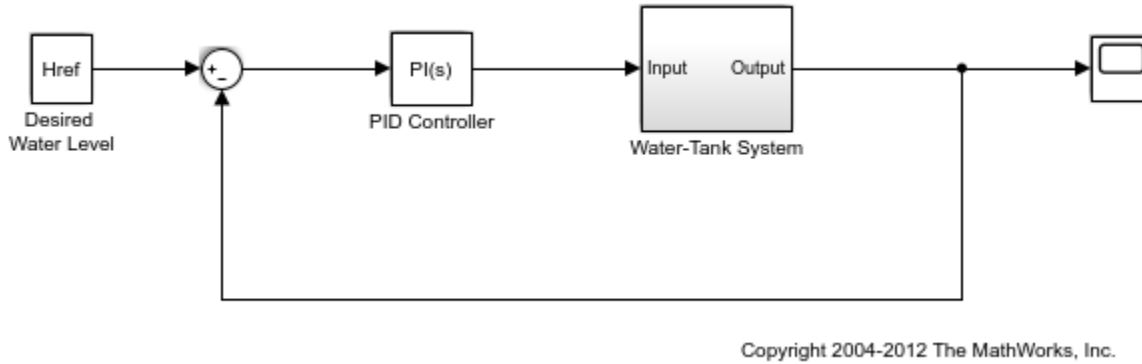
```
Sample time: 0.1 seconds
Discrete-time state-space model.
```

The linearized plant is a discrete-time state-space model with a sample time of `0.1`.

Linearize Model at Multiple Snapshot Times

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



To compute the closed-loop transfer function, first specify the linearization input and output signals.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

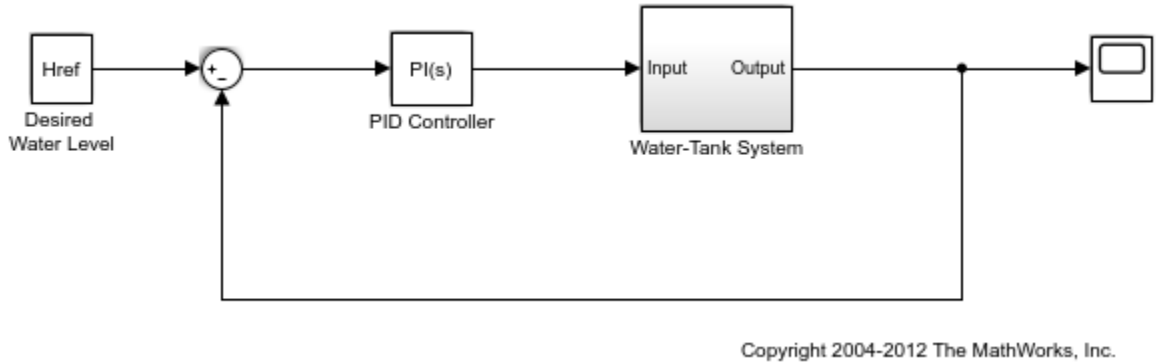
Simulate `sys` and linearize the model at `0` and `10` seconds. Return the operating points that correspond to these snapshot times; that is, the operating points at which the model was linearized.

```
[linsys,linop] = linearize(sys,io,[0,10]);
```

Batch Linearize Plant Model and Obtain Linearization Offsets

Open the Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Vary parameters `A` and `b` within 10% of their nominal values.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the model.

```
opspec = operspec(sys);
```

Trim the model using the specified operating point specification, parameter grid. Suppress the display of the operating point search report.

```
opt = findopOptions('DisplayReport','off');
[op,opreport] = findop(sys,opspec,params,opt);
```

`op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

Specify the block path for the plant model.

```
blockpath = 'watertank/Desired  Water Level';
```

To store offsets during linearization, create a linearization option set and set `StoreOffsets` to `true`.

```
options = linearizeOptions('StoreOffsets',true);
```

Batch linearize the plant at the trimmed operating points, using the specified I/O points and parameter variations.

```
[linsys,linop,info] = linearize(sys,blockpath,op,params,options);
```

You can use the offsets in `info.Offsets` when configuring an LPV System block.

```
info.Offsets
```

```
ans =

  3x4 struct array with fields:

    x
    dx
    u
    y
    StateName
    InputName
    OutputName
    Ts
```

Input Arguments

`sys` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set that contains inputs, outputs, and openings, specified as a linearization I/O object or a vector of linearization I/O objects. To create `io`:

- Define the inputs, outputs, and openings using `linio`.
- If the inputs, outputs, and openings are specified in the Simulink model, extract these points from the model using `getlinio`.

Each linearization I/O object in `io` must correspond to the Simulink model `sys` or some normal mode model reference in the model hierarchy.

If you omit `io`, then `linearize` uses the root level inports and outports of the model as analysis points.

For more information on specifying linearization inputs, outputs, and openings, see “Specify Portion of Model to Linearize” on page 2-13.

`op` — Operating point

operating point object | array of operating point objects

Operating point for linearization, specified as one of the following:

- Operating point object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of operating point objects, specifying multiple operating points. To create an array of operating point objects, you can:
  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-65.

If you specify parameter variations using `param`, and the parameters:

- Affect the model operating point, then specify `op` as an array of operating points with the same dimensions as the parameter value grid. To obtain the operating points that

correspond to the parameter value combinations, batch trim your model using `param` before linearization. For more information, see "Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations" on page 3-25.

- Do not affect the model operating point, then specify `op` as a single operating point.

`blockpath` — Block or subsystem

character vector | string

Block or subsystem to linearize, specified as a character vector or string that contains its full block path.

The software treats the inports and outports of the specified block as open-loop inputs and outputs, which isolates it from the rest of the model before linearization.

`blocksub` — Substitute linearizations for blocks and subsystems

structure | structure array

Substitute linearizations for blocks and subsystems, specified as a structure or an n-by-1 structure array, where n is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

To study the effects of varying the linearization of a block on the model dynamics, you can batch linearize your model by specifying multiple substitute linearizations for a block.

Each substitute linearization structure has the following fields:

`Name` — Block path

character vector | string

Block path of the block for which you want to specify the linearization, specified as a character vector or string.

`Value` — Substitute linearization

double | double array | LTI model | model array | structure

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.

- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.
- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:
  - Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
  - Specify `op` as an array of operating points and `Value` as a model array, the dimensions of `Value` must match the size of `op`.
  - Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for one or more of these blocks, the dimensions of the arrays must match.
- Structure with the following fields:

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following: • MATLAB expression • Name of a "Custom Linearization Function" on page 13-186 in your current working folder or on the MATLAB path The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.
`ParameterValues`	Linearization function parameter values, specified as a vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`param` — Parameter samples

structure | structure array

Parameter samples for linearization, specified as one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  params(2).Name = 'b';
  params(2).Value = b_grid;
  ```

For more information, see “Specify Parameter Samples for Batch Linearization” on page 3-62.

If `param` specifies tunable parameters only, the software batch linearizes the model using a single model compilation.

To compute the offsets required by the LPV System block, specify `param`, and set `options.StoreOffsets` to `true`. You can then return additional linearization information in `info`, and extract the offsets using `getOffsetsForLPV`.

`tsnapshot` — Simulation snapshot times

scalar | vector

Simulation snapshot times at which to linearize the model, specified as a scalar for a single snapshot, or a vector for multiple snapshots. The software simulates `sys` and linearizes the model at the specified snapshot times.

If you also specify parameter variations using `param`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`stateorder` — State order in linearization results
cell array of character vectors

State order in linearization results, specified as a cell array of block paths or state names. The order of the block paths and states in `stateorder` indicates the order of the states in `linsys`.

You can specify block paths for any blocks in `sys` that have states, or any named states in `sys`.

You do not have to specify every block and state from `sys` in `stateorder`. The states you specify appear first in `linsys`, followed by the remaining states in their default order.

`options` — Linearization algorithm options
`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` option set.

Output Arguments

`linsys` — Linearization result
state-space model | array of state-space models

Linearization result, returned as a state-space model or an array of state-space models. The dimensions of `linsys` depend on the specified parameter variations and block substitutions, and the operating points or snapshots at which you linearize the model.

Note If you specify more than one of `op`, `param`, or `blocksub.Value` as an array, then their dimensions must match.

Parameter Variation	Block Substitution	Linearize at...	Resulting `linsys` Dimensions
No parameter variation	No block substitution	Model operating point	Single state-space model
		Single operating point, specified by `op`	
		Single snapshot, specified by `tsnapshot`	
		N_1-by-...-by-N_m array of operating points, specified by `op`	N_1-by-...-by-N_m
		N_s snapshots, specified by `tsnapshot`	Column vector of length N_s
	N_1-by-...-by-N_m model array for at least one block, specified by `blocksub.Value`	Model operating point	N_1-by-...-by-N_m
		Single operating point, specified by `op`	
		Single snapshot, specified by `tsnapshot`	
		N_1-by-...-by-N_m array of operating points, specified by `op`	
		N_s snapshots, specified by `tsnapshot`	N_s-by-N_1-by-...-by-N_m
N_1-by-...-by-N_m parameter grid, specified by `param`	Either no block substitution or an N_1-by-...-by-N_m model array for at least one block,	Model operating point	N_1-by-...-by-N_m
		Single operating point, specified by `op`	

Parameter Variation	Block Substitution	Linearize at...	Resulting `linsys` Dimensions
	specified by `blocksub.Value`	Single snapshot, specified by `tsnapshot`	
		N_1-by-. . .-by-N_m array of operating points, specified by `op`	
		N_s snapshots, specified by `tsnapshot`	N_s-by-N_1-by-. . .-by-N_m

For example, suppose:

- `op` is a 4-by-3 array of operating point objects and you do not specify parameter variations or block substitutions. In this case, `linsys` is a 4-by-3 model array.
- `op` is a single operating point object and `param` specifies a 3-by-4-by-2 parameter grid. In this case, `linsys` is a 3-by-4-by-2 model array.
- `tsnapshot` is a row vector with two elements and you do not specify `param`. In this case, `linsys` is a column vector with two elements.
- `tsnapshot` is a column vector with three elements and `param` specifies a 5-by-6 parameter grid. In this case, `linsys` is a 3-by-5-by-6 model array.
- `op` is a single operating point object, you do not specify parameter variations, and `blocksub.Value` is a 2-by-3 model array for one block in the model. In this case, `linsys` is a 2-by-3 model array.
- `tsnapshot` is a column vector with four elements, you do not specify parameter variations, and `blocksub.Value` is a 1-by-2 model array for one block in the model. In this case, `linsys` is a 4-by-1-by-2 model array.

For more information on model arrays, see “Model Arrays” (Control System Toolbox).

`linop` — Operating point

operating point object | array of operating point objects

Operating point at which the model was linearized, returned as an operating point object or an array of operating point objects with the same dimensions as `linsys`. Each

element of `linop` is the operating point at which the corresponding `linsys` model was obtained.

If you specify an operating point or operating point array using `op`, then `linop` is a copy of `op`. If you specify `op` as a single operating point and also specify parameter variations using `param`, then `linop` is an array with the same dimensions as the parameter grid. In this case, the elements of `linop` are scalar expanded copies of `op`.

To determine whether the model was linearized at a reasonable operating point, view the states and inputs in `linop`.

`info` — Linearization information

structure

Linearization information, returned as a structure with the following fields:

`Offsets` — Linearization offsets

`[]` (default) | structure | structure array

Linearization offsets that correspond to the operating point at which the model was linearized, returned as `[]` if `options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `linsys` is a single state-space model, then `Offsets` is a structure.
- If `linsys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `linsys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `linsys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `linsys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `linsys`.
`dx`	Derivative offsets for continuous time systems, or updated state values for discrete-time systems, returned as a column vector of length n_x.

Field	Description
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `linsys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `linsys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `linsys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `sys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see "Approximating Nonlinear Behavior Using an Array of LTI Systems" on page 3-91.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Definitions

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>BlockName</code></td><td>Name of the block for which you are specifying a custom linearization.</td></tr>
<tr><td><code>Parameters</code></td><td>Block parameter values, specified as a structure array with <code>Name</code> and <code>Value</code> fields. <code>Parameters</code> contains the names and values of the parameters you specify in the <code>blocksub.Value.ParameterNames</code> and <code>blocksub.Value.ParameterValues</code> fields.</td></tr>
<tr><td><code>Inputs</code></td><td>Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in <code>Inputs</code> has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>BlockName</code></td><td>Full block path of the block whose output connects to the corresponding block input.</td></tr>
<tr><td><code>PortIndex</code></td><td>Output port of the block specified by <code>BlockName</code> that connects to the corresponding block input.</td></tr>
<tr><td><code>Values</code></td><td>Value of the signal specified by <code>BlockName</code> and <code>PortIndex</code>. If this signal is a vector signal, then <code>Values</code> is a vector with the same dimension.</td></tr>
</table></td></tr>
<tr><td><code>ny</code></td><td>Number of output channels of the block linearization.</td></tr>
<tr><td><code>nu</code></td><td>Number of input channels of the block linearization.</td></tr>
<tr><td><code>BlockLinearization</code></td><td>Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using <code>BlockLinearization</code>.</td></tr>
</table>

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

Algorithms

Model Properties for Linearization

By default, `linearize` automatically sets the following Simulink model properties:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

- `SaveFormat = 'StructureWithTime'`

After linearization, Simulink restores the original model properties.

Block-by-Block Linearization

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations.

The software determines the input and state levels for each block from the operating point, and requests the Jacobian for these levels from each block.

For some blocks, the software cannot compute an analytical linearization. For example:

- Some nonlinearities do not have a defined Jacobian.
- Some discrete blocks, such as state charts and triggered subsystems, tend to linearize to zero.
- Some blocks do not implement a Jacobian.
- Custom blocks, such as S-Function blocks and MATLAB Function blocks, do not have analytical Jacobians.

You can specify a custom linearization for any such blocks for which you know the expected linearization. If you do not specify a custom linearization, the software linearizes the model by perturbing the block inputs and states and measuring the response to these perturbations. For each input and state, the default perturbation level

is $10^{-5}\left(1+|x|\right)$, where x is the value of the corresponding input or state at the operating point. For information on how to change perturbation levels for individual blocks, see “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-181.

For more information, see “Linearize Nonlinear Models” on page 2-3 and “Exact Linearization Algorithm” on page 2-207

Full-Model Numerical Perturbation

You can also linearize your system using full-model numerical perturbation. To do so, create a `linearizeOptions` set and set the `LinearizationAlgorithm` to `'numericalpert'`. The software computes the linearization of the full model by

perturbing the values of the root-level inports and states of the model. This algorithm ignores linear analysis points set in the model and uses root-level inports and outports instead.

For each inport and state, the software perturbs the model and measures the model response to these perturbations. You can configure the state and input perturbation levels using the `NumericalPertRel` linearization options.

Block-by-block linearization has several advantages over full-model numerical perturbation:

- Many Simulink blocks have a preprogrammed linearization that provides an exact linearization of the block.
- You can use linear analysis points to specify a portion of the model to linearize.
- You can configure blocks to use custom linearizations without affecting your model simulation.
- Structurally nonminimal states are automatically removed.
- You can specify linearizations that include uncertainty (requires Robust Control Toolbox software).
- You can obtain detailed diagnostic information
- When linearizing multirate models, you can use different rate conversion methods. Full-model numerical perturbation can only use zero-order-hold rate conversion.

For more information, see “Linearize Nonlinear Models” on page 2-3 and “Exact Linearization Algorithm” on page 2-207.

Alternatives

As an alternative to the `linearize` function, you can linearize models using one of the following methods:

- To interactively linearize models, use the **Linear Analysis Tool**. For an example, see “Linearize Simulink Model at Model Operating Point” on page 2-69.
- To obtain multiple transfer functions without modifying the model or creating an analysis point set for each transfer function, use an `slLinearizer` interface. For an example, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32.

Although both Simulink Control Design software and the Simulink `linmod` function perform block-by-block linearization, Simulink Control Design linearization functionality has a more flexible user interface and uses Control System Toolbox numerical algorithms. For more information, see “Linearization Using Simulink Control Design Versus Simulink” on page 2-11.

See Also

Linear Analysis Tool | `findop` | `linearizeOptions` | `slLinearizer`

Topics

“Linearize Simulink Model at Model Operating Point” on page 2-69
“Linearize at Trimmed Operating Point” on page 2-85
“Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-20

Introduced in R2006a

linearizeOptions

Set linearization options

Syntax

```
options = linearizeOptions
options = linearizeOptions(Name,Value)
```

Description

`options = linearizeOptions` returns the default linearization option set.

`options = linearizeOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Option Set for Linearization

Create a linearization option set that sets the rate conversion method to the Tustin method with prewarping at a frequency of 10 rad/s. Additionally, instruct the linearization not to omit blocks outside the linearization path.

```
options = linearizeOptions('RateConversionMethod','prewarp',...
                           'PreWarpFreq',10,...
                           'BlockReduction','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = linearizeOptions;
options.RateConversionMethod = 'prewarp';
```

```
options.PreWarpFreq = 10;
options.BlockReduction = 'off';
```

Input Arguments

Name-Value Pair Arguments

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

Example: `'RateConversionMethod','prewarp'` sets the rate conversion method to the Tustin method with prewarping.

`LinearizationAlgorithm` — Algorithm used for linearization

`'blockbyblock'` (default) | `'numericalpert'`

Algorithm used for linearization, specified as the comma-separated pair consisting of `'LinearizationAlgorithm'` and one of the following:

- `'blockbyblock'` — Individually linearize each block in the model, and combine the results to produce the linearization of the specified system.
- `'numericalpert'` — Full-model numerical-perturbation linearization in which root-level inports and states are numerically perturbed. This algorithm ignores linear analysis points set in the model and uses root-level inports and outports instead.

Block-by-block linearization has several advantages over full-model numerical perturbation:

- Many Simulink blocks have a preprogrammed linearization that provides an exact linearization of the block.
- You can use linear analysis points to specify a portion of the model to linearize.
- You can configure blocks to use custom linearizations without affecting your model simulation.
- Structurally nonminimal states are automatically removed.
- You can specify linearizations that include uncertainty (requires Robust Control Toolbox software).

- You can obtain detailed diagnostic information

`SampleTime` — Sample time of linearization result
`-1` (default) | `0` | positive scalar

Sample time of linearization result, specified as the comma-separated pair consisting of `'SampleTime'` and one of the following:

- `-1` — Use the longest sample time that contributes to the linearized model.
- `0` — Use for continuous-time systems.
- Positive scalar — Specify the sample time for discrete-time systems.

`UseFullBlockNameLabels` — Flag indicating whether to truncate names of I/Os and states
`'off'` (default) | `'on'`

Flag indicating whether to truncate names of I/Os and states in the linearized model, specified as the comma-separated pair consisting of `'UseFullBlockNameLabels'` and either:

- `'off'` — Use truncated names for the I/Os and states in the linearized model.
- `'on'` — Use the full block path to name the I/Os and states in the linearized model.

`UseBusSignalLabels` — Flag indicating whether to use bus signal channel numbers or names
`'off'` (default) | `'on'`

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as the comma-separated pair consisting of `'UseBusSignalLabels'` and one of the following:

- `'off'` — Use bus signal channel numbers to label I/Os on bus signals in the linearized model.
- `'on'` — Use bus signal names to label I/Os on bus signals in the linearized model. Bus signal names appear in the results when the I/O points are located at the output of the following blocks:
  - Root-level inport block containing a bus object
  - Bus creator block

- Subsystem block whose source traces back to the output of a bus creator block
- Subsystem block whose source traces back to a root-level inport by passing through only virtual or nonvirtual subsystem boundaries

`StoreOffsets` — Flag indicating whether to compute linearization offsets
`false` (default) | `true`

Flag indicating whether to compute linearization offsets for inputs, outputs, states, and state derivatives or updated states, specified as the comma-separated pair consisting of `'StoreOffsets'` and one of the following:

- `false` — Do not compute linearization offsets.
- `true` — Compute linearization offsets.

You can configure an LPV System block using linearization offsets. For an example, see “Approximating Nonlinear Behavior Using an Array of LTI Systems” on page 3-91

`StoreAdvisor` — Flag indicating whether to store diagnostic information
`false` (default) | `true`

Flag indicating whether to store diagnostic information during linearization, specified as the comma-separated pair consisting of `'StoreAdvisor'` and one of the following:

- `false` — Do not store linearization diagnostic information.
- `true` — Store linearization diagnostic information.

Linearization commands store and return diagnostic information in a `LinearizationAdvisor` object. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-40.

`BlockReduction` — Flag indicating whether to omit blocks that are not on the linearization path
`'on'` (default) | `'off'`

Flag indicating whether to omit blocks that are not in the linearization path, specified as the comma-separated pair consisting of `'BlockReduction'` and one of the following:

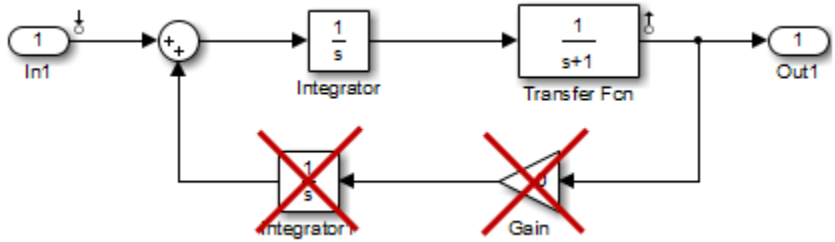
- `'on'` — Return a linearized model that does not include states from noncontributing linearization paths.

- `'off'` — Return a linearized model that includes all the states of the model.

Dead linearization paths can include:

- Blocks that linearize to zero.
- Switch blocks that are not active along the path.
- Disabled subsystems.
- Signals marked as open-loop linearization points.

For example, if this flag set to `'on'`, the linearization result of the model shown in the following figure includes only two states. It does not include states from the two blocks outside the linearization path. These states do not appear because these blocks are on a dead linearization path with a block that linearizes to zero (the zero gain block).



This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`. `BlockReduction` is always treated as `'on'` when `LinearizationAlgorithm` is `'numericalpert'`.

`IgnoreDiscreteStates` — Flag indicating whether to remove discrete-time states
`'off'` (default) | `'on'`

Flag indicating whether to remove discrete-time states from the linearization, specified as the comma-separated pair consisting of `'IgnoreDiscreteStates'` and one of the following:

- `'off'` — Always include discrete-time states.
- `'on'` — Remove discrete states from the linearization. Use this option when performing continuous-time linearization (`SampleTime = 0`) to accept the `D` value for all blocks with discrete-time states.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`RateConversionMethod` — Rate conversion method

`'zoh'` (default) | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`

Method used for rate conversion when linearizing a multirate system, specified as the comma-separated pair consisting of `'RateConversionMethod'` and one of the following:

- `'zoh'` — Zero-order hold rate conversion method
- `'tustin'` — Tustin (bilinear) method
- `'prewarp'` — Tustin method with frequency prewarp. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.
- `'upsampling_zoh'` — Upsample discrete states when possible, and use `'zoh'` otherwise.
- `'upsampling_tustin'` — Upsample discrete states when possible, and use `'tustin'` otherwise.
- `'upsampling_prewarp'` — Upsample discrete states when possible, and use `'prewarp'` otherwise. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.

For more information on rate conversion and linearization of multirate models, see:

- “Linearization of Multirate Models”.
- “Linearization Using Different Rate Conversion Methods”.
- “Continuous-Discrete Conversion Methods” (Control System Toolbox) .

Note If you use a rate conversion method other than `'zoh'`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`PreWarpFreq` — Prewarp frequency

`0` (default) | positive scalar

Prewarp frequency in rad/s, specified as the comma-separated pair consisting of `'PreWarpFreq'` and a nonnegative scalar. This option applies only when `RateConversionMethod` is either `'prewarp'` or `'upsampling_prewarp'`.

`UseExactDelayModel` — Flag indicating whether to compute linearization with exact delays
`'off'` (default) | `'on'`

Flag indicating whether to compute linearization with exact delays, specified as the comma-separated pair consisting of `'UseExactDelayModel'` and one of the following:

- `'off'` — Return a linear model with approximate delays.
- `'on'` — Return a linear model with exact delays.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values
`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for linearization, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.
- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

For more information about model compilation when you linearize with parameter variation, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10.

`NumericalPertRel` — Numerical perturbation level
`1e-5` (default) | positive scalar

Numerical perturbation level, specified as the comma-separated pair consisting of `'NumericalPertRel'` and a positive scalar. This option applies only when `LinearizationAlgorithm` is `'numericalpert'`.

The perturbation levels for the system states are:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |x|$$

The perturbation levels for the system inputs are:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |u|$$

You can override these values using the `NumericalXPert` or `NumericalUPert` options.

`NumericalXPert` — State perturbation levels

`[]` (default) | operating point object

State perturbation levels, specified as the comma-separated pair consisting of `'NumericalXPert'` and an operating point object. This option applies only when `LinearizationAlgorithm` is `'numericalpert'`.

To set individual perturbation levels for each state:

1. Create an operating point object for the model using the `operpoint` command.

   ```
   xPert = operpoint('watertank');
   ```

2. Set the state values in the operating point object to the perturbation levels.

   ```
   xPert.States(1).x = 2e-3;
   xPert.States(2).x = 3e-3;
   ```

3. Set the value of the `NumericalXPert` option to the operating point object.

   ```
   opt = linearizeOptions('LinearizationAlgorithm','numericalpert');
   opt.NumericalXPert = xPert;
   ```

If `NumericalXPert` is empty, `[]`, the linearization algorithm derives the state perturbation levels using `NumericalPertRel`.

`NumericalUPert` — Input perturbation levels

`[]` (default) | operating point object

Input perturbation levels, specified as the comma-separated pair consisting of `'NumericalUPert'` and an operating point object. This option applies only when `LinearizationAlgorithm` is `'numericalpert'`.

To set individual perturbation levels for each input:

1. Create an operating point object for the model using the `operpoint` command.

   ```
   uPert = operpoint('watertank');
   ```

2. Set the input values in the operating point object to the perturbation levels.

   ```
   uPert.Inputs(1).x = 3e-3;
   ```

3. Set the value of the `NumericalUPert` option to the operating point object.

   ```
   opt = linearizeOptions('LinearizationAlgorithm','numericalpert');
   opt.NumericalUPert = uPert;
   ```

If `NumericalUPert` is empty, `[]`, the linearization algorithm derives the input perturbation levels using `NumericalPertRel`.

Output Arguments

`options` — Linearization options

`linearizeOptions` option set

Linearization options, returned as a `linearizeOptions` option set.

See Also

`linearize` | `linlft` | `slLinearizer` | `ulinearize`

Introduced in R2013b

linio

Create linear analysis point for Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
io = linio(block,port)
io = linio(block,port,type)
io = linio(block,port,type,[],busElement)
```

Description

`io = linio(block,port)` creates a linearization I/O object that represents an input perturbation analysis point for the signal that originates from the specified output `port` of a Simulink `block`.

`io = linio(block,port,type)` creates an analysis point of the specified `type`.

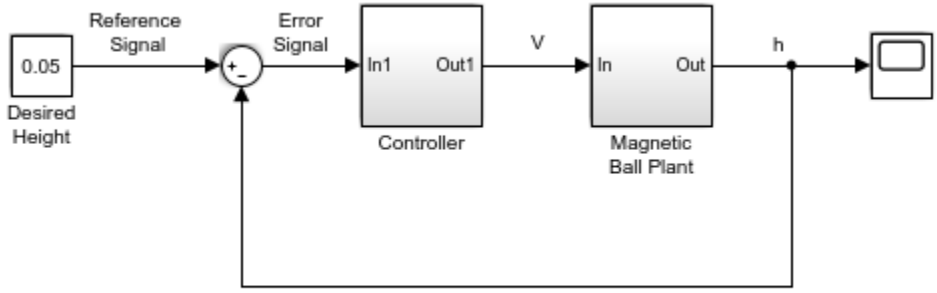
`io = linio(block,port,type,[],busElement)` creates an analysis point for an element of a bus signal.

Examples

Create Analysis Points for Simulink Model

Open Simulink model.

```
open_system('magball')
```



To specify multiple analysis points for linearization, create a vector of linearization I/O objects.

Create an input perturbation analysis point at the output port of the Controller block.

```
io(1) = linio('magball/Controller',1);
```

Create an open-loop output analysis point at the output of the Magnetic Ball Plant block. An open-loop output point is an output measurement followed by a loop opening.

```
io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
```

View the specified analysis points.

```
io
```

```
1x2 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: magball/Controller
- Port: 1
2. Linearization open-loop output located at the following signal:
- Block: magball/Magnetic Ball Plant
- Port: 1
```

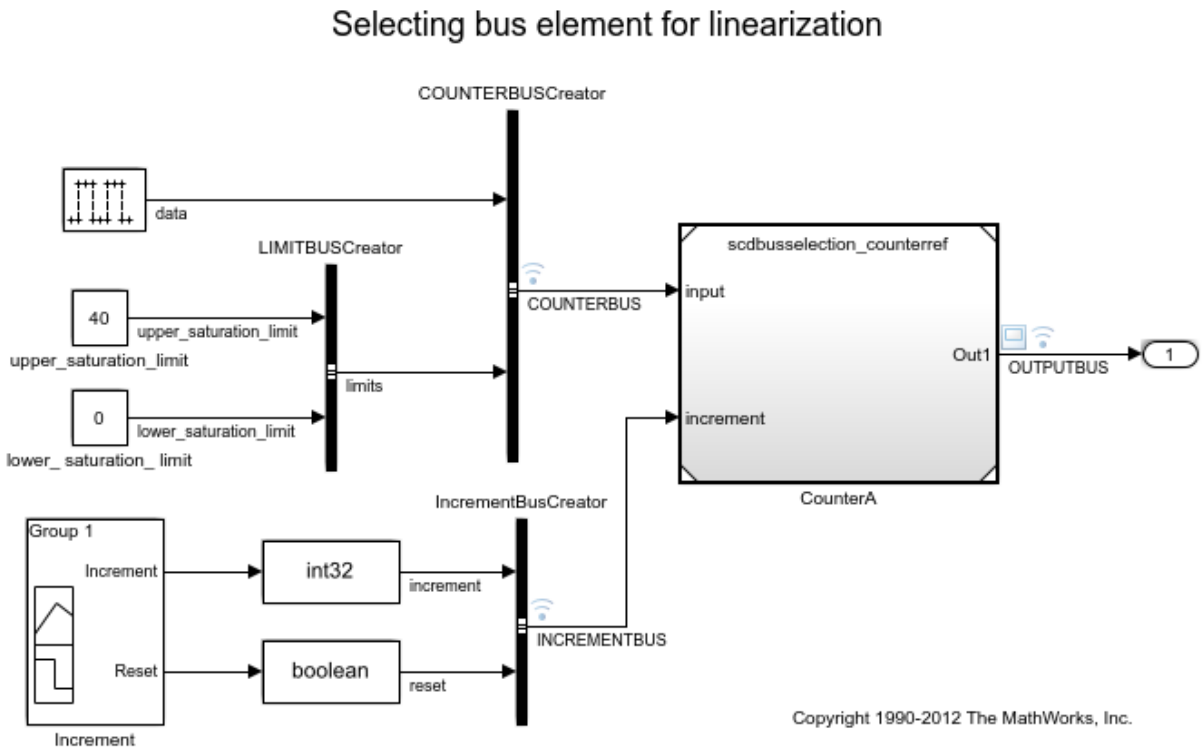
You can use these analysis points to linearize only the Magnetic Ball Plant subsystem. To do so, pass `io` to the `linearize` command or to an `slLinearizer` interface.

Unlike specfying analysis points directly in the Simulink model, when you create analysis points using `linio`, no annotations are added to the model.

Select Individual Bus Element as Analysis Point

Open Simulink model.

```
mdl = 'scdbusselection';
open_system(mdl)
```



The `COUNTERBUS` signal, which originates from the CounterBusCreator block, contains multiple bus elements.

Specify the `upper_saturation_limit` bus element as a linearization input. Select this element using dot notation, since it is within the nested `limits` bus.

```
io = linio('scdbusselection/COUNTERBUSCreator',1,'input',[],...
           'limits.upper_saturation_limit');
```

Input Arguments

`block` — Simulink block

character vector | string

Simulink block from which the analysis point originates, specified as a character vector or string that contains its full block path. For example, to mark an analysis point at an output of the Controller block in the `magball` model, specify `block` as `'magball/Controller'`.

`port` — Output port

positive integer

Output port of `block` from which the analysis point originates, specified as a positive integer.

`port` must be a valid port number for the specified `block`.

`type` — Analysis point type

`'input'` (default) | `'output'` | `'loopbreak'` | ...

Analysis point type, specified as one of the following:

- `'input'` — Input perturbation
- `'output'` — Output measurement
- `'loopbreak'` — Loop break
- `'openinput'` — Open-loop input
- `'openoutput'` — Open-loop output
- `'looptransfer'` — Loop transfer
- `'sensitivity'` — Sensitivity
- `'compsensitivity'` — Complementary sensitivity

For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-13.

`busElement` — Bus element name
character vector | string

Bus element name, specified as a character vector or string. When adding elements within a nested bus structure, use dot notation to access the elements of the nested bus. For an example, see “Select Individual Bus Element as Analysis Point” on page 13-202.

Output Arguments

`io` — Analysis point
linearization I/O object

Analysis point, returned as a linearization I/O object. Use `io` to specify a linearization input, output, or loop opening when using the `linearize` command. For more information, see “Specify Portion of Model to Linearize” on page 2-13.

Each linearization I/O object has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.

Property	Description
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-13.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

Alternative Functionality

Linear Analysis Tool

You can interactively configure analysis points using the Linear Analysis Tool. For more information see, “Specify Portion of Model to Linearize in Linear Analysis Tool” on page 2-29.

Simulink Model

You can also specify analysis points directly in a Simulink model. When you do so, the analysis points are saved within the model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-21.

`slLinearizer` and `slTuner` Interfaces

If you want to obtain multiple open-loop or closed-loop transfer functions from the linearized system without recompiling the model, you can specify linear analysis points using an `slLinearizer` interface. For more information, see “Mark Signals of Interest for Batch Linearization” on page 3-13. Similarly, if you want to tune a control system and obtain multiple open-loop or closed-loop transfer functions from the resulting system, you can specify linear analysis points using an `slTuner` interface. For more information, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48.

See Also

`getlinio` | `linearize` | `setlinio`

Topics

“Specify Portion of Model to Linearize” on page 2-13

Introduced before R2006a

linlft

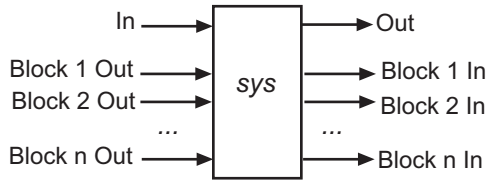
Linearize model while removing contribution of specified blocks

Syntax

```
lin_fixed = linlft(sys,io,blocks)
[lin_fixed,lin_blocks] = linlft( ___ )
[ ___ ] = linlft( ___ ,opt)
```

Description

`lin_fixed = linlft(sys,io,blocks)` linearizes the Simulink model named `sys` while removing the contribution of certain blocks. Specify `sys` as a character vector or string. Specify the full block path of the blocks to ignore in the cell array of character vectors or string array called `blocks`. The linearization occurs at the operating point specified in the Simulink model, which includes the ignored blocks. You can optionally specify linearization points (linear analysis points) in the I/O object `io`. The resulting linear model `lin_fixed` has this form:



The top channels In and Out correspond to the linearization points you specify in the I/O object `io`. The remaining channels correspond to the connection to the ignored blocks.

When you use `linlft` and specify the `'block-by-block'` linearization algorithm in `linearizeOptions`, you can use all the variations of the input arguments for `linearize`.

You can linearize the ignored blocks separately using `linearize`, and then combine the linearization results using `linlftfold`.

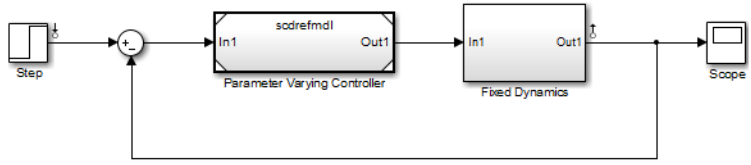
`[lin_fixed,lin_blocks] = linlft( ___ )` returns the linearizations for each of the blocks specified in `blocks`. If `blocks` contains a single block path, `lin_blocks` is a single state-space (`ss`) model. If `blocks` is an array identifying multiple blocks, `lin_blocks` is a cell array of state-space models. The full block path for each block in `lin_blocks` is stored in the `Notes` property of the state-space model.

`[ ___ ] = linlft( ___ ,opt)` uses additional linearization options, specified as a `linearizeOptions` option set.

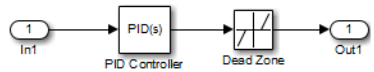
Examples

Linearize the following parts of the `scdtopmdl` Simulink model separately, and then combine the results:

- Fixed portion, which contains everything except the Parameter Varying Controller model reference



- Parameter Varying Controller model reference, which references the `scdrefmdl` model




```
% Open the Simulink model
topmdl = 'scdtopmdl';

% Linearize the model without the Parameter Varying Controller
io = getlinio(topmdl);
blocks = {'scdtopmdl/Parameter Varying Controller'};
sys_fixed = linlft(topmdl,io,blocks);

% Linearize the Parameter Varying Controller
refmdl = 'scdrefmdl';
sys_pv = linearize(refmdl);
```

```
% Combine the results
BlockSubs(1) = struct('Name',blocks{1},'Value',sys_pv);
sys_fold = linlftfold(sys_fixed,BlockSubs);
```

See Also

getlinio | linearize | linearizeOptions | linio | linlftfold | operpoint

Introduced in R2009b

linlftfold

Combine linearization results from specified blocks and model

Syntax

```
lin = linlftfold(lin_fixed,blocksubs)
```

Description

`lin = linlftfold(lin_fixed,blocksubs)` combines the following linearization results into one linear model `lin`:

- Linear model `lin_fixed`, which does not include the contribution of specified blocks in your Simulink model.

  Compute `lin_fixed` using `linlft`.
- Block linearizations for the blocks excluded from `lin_fixed`

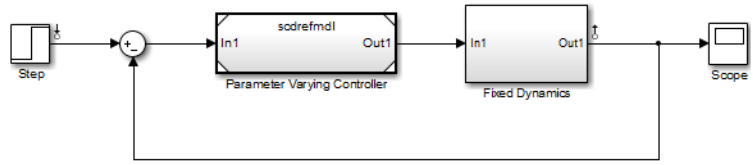
  You specify the block linearizations in a structure array `blocksubs`, which contains two fields:

  - `'Name'` is a character vector or string specifying the block path of the Simulink block to replace.
  - `'Value'` is the value of the linearization for each block.

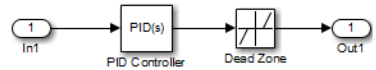
Examples

Linearize the following parts of the `scdtopmdl` Simulink model separately and then combine the results:

- Fixed portion, which contains everything except the Parameter Varying Controller model reference



- Parameter Varying Controller model reference, which references the `scdrefmdl` model




```
% Open the Simulink model
topmdl = 'scdtopmdl';

% Linearize the model without the Parameter Varying Controller
io = getlinio(topmdl);
blocks = {'scdtopmdl/Parameter Varying Controller'};
sys_fixed = linlft(topmdl,io,blocks);

% Linearize the Parameter Varying Controller
refmdl = 'scdrefmdl';
sys_pv = linearize(refmdl);

% Combine the results
BlockSubs(1) = struct('Name',blocks{1},'Value',sys_pv);
sys_fold = linlftfold(sys_fixed,BlockSubs);
```

See Also

`getlinio` | `linearize` | `linio` | `linlft` | `operpoint`

Introduced in R2009b

operpoint

Create operating point for Simulink model

Syntax

```
op = operpoint(sys)
```

Description

`op = operpoint(sys)` returns an object, `op`, containing the operating point of a Simulink model, `sys`. Specify `sys` as either a character vector or a string. Use the object with the function `linearize` to create linearized models. The operating point object has the following properties:

- `Model` — Simulink model name, specified as a character vector.
- `States` — State operating point specification, specified as a structure array. Each structure in the array represents the supported states of one Simulink block. (For a list of supported states for operating point objects, see "Simulink Model States Included in Operating Point Object" on page 1-4.) Edit the properties of this object using dot notation or the `set` function.

  Each `States` structure has the following fields:

`Nx`(read only)	Number of states in the Simulink block.
`Block`	Simulink block name.
`StateName`	Name of state, specified as a character vector.
`x`	Simulink block state values, specified as a vector of states. This vector includes all supported states. When creating state value specifications for operating point searches using `findop` and you set the value of a state that you want fixed, also set the `Known` field of the `States` property for that state to `1`.

`Ts`	(Only for discrete-time states) Sample time and offset of each Simulink block state, specified as a vector.
`SampleType`	State time rate, specified as one of the following values: • `'CSTATE'` — Continuous-time state • `'DSTATE'` — Discrete—time state
`inReferencedModel`	Vector indicating whether each state is inside a reference model: • `1` — State is inside a reference model • `0` — State is in the current model file
`Description`	Block state description, specified as a character vector.

- `Inputs` — Input level at the operating point, specified as a vector of input objects. Each input object represents the input levels of one root-level inport block in the Simulink block.

  Each entry in `Inputs` has the following fields:

`Block`	Inport block name.
`PortWidth`	Number of inport block signals.
`PortDimensions`	Dimension of signals accepted by the inport.
`u`	Inport block input levels at the operating point, specified as a vector of input levels. When creating input specifications for operating-point searches using `findop`, also set the `Known` field of the `Inputs` property for known input levels that remain fixed during operating point search.
`Description`	Inport block input description, specified as a character vector.

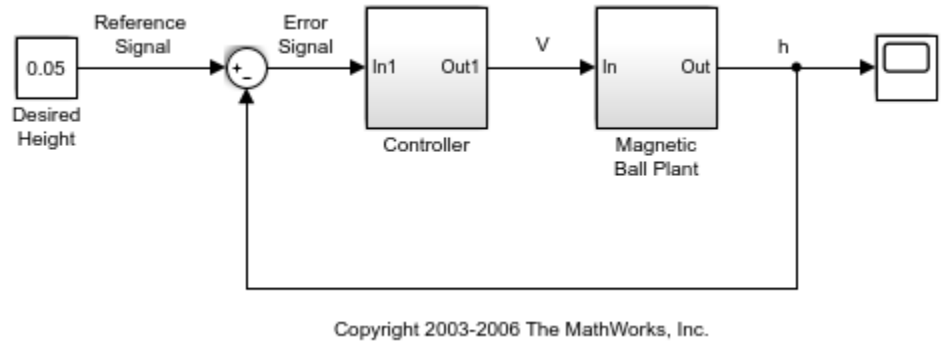
- `Time` — Times at which any time-varying functions in the model are evaluated, specified as a vector.
- `Version` — Object version number.

Examples

Create Operating Point for Simulink Model

Open Simulink model.

```
open_system('magball')
```



Create operating point for the model.

```
op = operpoint('magball')
```

```
 Operating point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
```

```
Inputs: None
----------
```

`op` lists each block in the model that has states. There are no root-level inports in this model, therefore `op` does not contain inputs.

Alternatives

As an alternative to the `operpoint` function, create operating points in the Linear Analysis Tool. See "Compute Steady-State Operating Point from State Specifications" on page 1-14 and "Compute Steady-State Operating Point from Output Specifications" on page 1-28.

See Also

`get` | `linearize` | `operspec` | `set` | `update`

Introduced before R2006a

operspec

Operating point specifications

Syntax

```
opspec = operspec(mdl)
opspec = operspec(mdl,dim)
```

Description

`opspec = operspec(mdl)` returns the default operating point specification object for the Simulink model `mdl`. Use `opspec` for steady-state operating point trimming using `findop`.

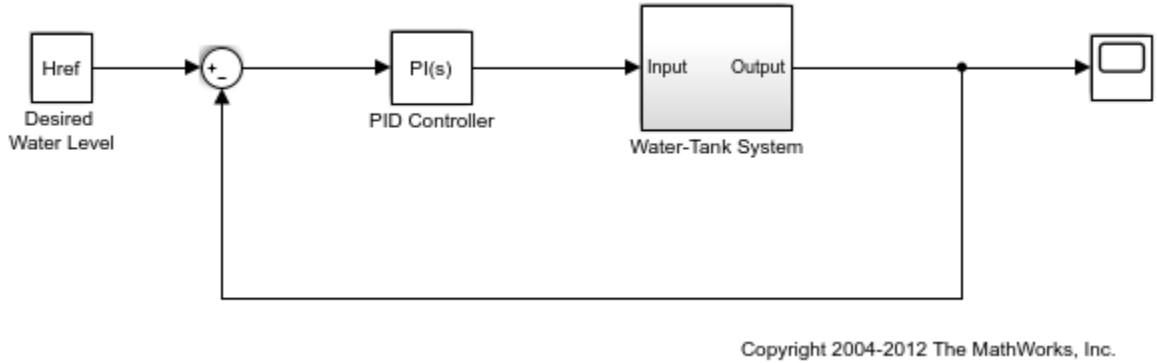
`opspec = operspec(mdl,dim)` returns an array of default operating point specification objects with the specified dimensions, `dim`.

Examples

Create Operating Point Specification Object

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create the default operating point specification object for the model.

```
opspec = operspec(sys)
```

```
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) watertank/PID Controller/Integrator
     spec:  dx = 0,  initial guess: 0
(2.) watertank/Water-Tank System/H
     spec:  dx = 0,  initial guess: 1

Inputs: None
----------

Outputs: None
----------
```

`opspec` contains specifications for the two states in the model. Since the model has no root level inports or outports, `opspec` does not contain input or output specifications. To add output specifications, use `addoutputspec`.

Modify the operating point specifications for each state using dot notation. For example, configure the first state to:

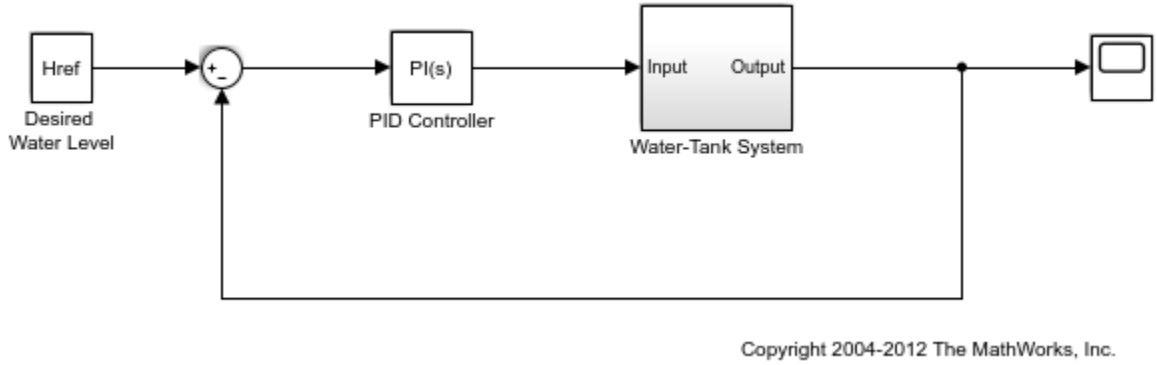
- Be at steady state.
- Have a lower bound of `0`.
- Have an initial value of `2` for trimming.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Create Array of Operating Point Specification Objects

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create a 2-by-3 array of operating point specification objects. You can batch trim model at multiple operating points using such arrays.

```
opspec = operspec(sys,[2,3]);
```

Each element of `opspec` contains a default operating point specification object for the model.

Modify the operating point specification objects using dot notation. For example, configure the second state of the specification object in row `1`, column `3`.

```
opspec(1,3).States(2).SteadyState = 1;
opspec(1,3).States(1).x = 2;
```

You can also create multidimensional arrays of operating point specification objects. For example, create a 3-by-4-by-5 array.

```
opspec = operspec(sys,[3,4,5]);
```

Input Arguments

`mdl` — Simulink model

character vector | string

Simulink model name, specified as a character vector or string.

`dim` — Array dimensions

integer | row vector of integers

Array dimensions, specified as one of the following:

- Integer — Create a column vector of `dim` operating point specification objects.
- Row vector of integers — Create an array of operating point specification objects with the dimensions specified by `dim`.

  For example, to create a 4-by-5 array of operating point specification objects, use:

  ```
  opspec = operspec(mdl,[4,5]);
  ```

  To create a multidimensional array of operating point specification objects, specify additional dimensions. For example, to create a 2-by-3-by-4 array, use:

  ```
  opspec = operspec(mdl,[2,3,4]);
  ```

Output Arguments

`opspec` — Operating point specifications

operating point specification object | array of operating point specification objects

Operating point specifications, returned as an operating point specification object or an array of such objects.

You can modify the operating point specifications using dot notation. For example, if `opspec` is a single operating point specification object, `opspec.States(1).x` accesses the state values of the first model state. If `opspec` is an array of specification objects `opspec(2,3).Inputs(1).u` accesses the input level of the first inport block for the specification in row `2`, column `3`.

Each specification object has the following properties:

Property	Description
`Model`	Simulink model name, returned as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>States</td><td>State operating point specifications, returned as a vector of state specification objects. Each entry in States represents the supported states of one Simulink block.

For a list of supported states for operating point objects, see “Simulink Model States Included in Operating Point Object” on page 1-4. Edit the properties of this object using dot notation or the set function.

Note If the block has multiple named continuous states, States contains one structure for each named state.

Each state specification object has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nx (read-only)</td><td>Number of states in the block</td></tr>
<tr><td>Block</td><td>Block path, returned as a character vector.</td></tr>
<tr><td>StateName</td><td>State name</td></tr>
<tr><td>x</td><td>Values of all supported block states, specified as a vector of length Nx.

If the corresponding flag in Known field of States is 1, x contains the known state values. Otherwise, x contains initial guesses for the state values.</td></tr>
<tr><td>Ts</td><td>(Only for discrete-time states) Sample time and offset of each supported block state, returned as a vector.</td></tr>
<tr><td>SampleType</td><td>State time rate, returned as one of the following:

• 'CSTATE' — Continuous-time state
• 'DSTATE' — Discrete-time state</td></tr>
<tr><td>inReferencedModel</td><td>Flag indicating whether the block is inside a reference model, returned as one of the following:

• 1 — Block is inside a reference model.</td></tr>
</table>
</td></tr>
</table>

Property	Description	
	Field	Description
		• `0` — Block is in the current model file.
	`Known`	Flags indicating whether state values are known during trimming, specified as a logical vector of length `Nx`. • `1` — Known value that is fixed during operating point search • `0` (default) — Unknown value found by optimization To fix a state during an operating point search, set the corresponding `Known` flag to `1`, and specify the value for that state using the `x` property of `States`.
	`SteadyState`	Flags indicating whether output values are at steady state during trimming, specified as a logical vector of length `Nx`. • `1` (default) — Equilibrium state • `0` — Nonequilibrium state
	`Min`	Minimum bounds on state values, specified as a vector of length `Nx`. By default, the minimum bound for each state is `-Inf`.
	`Max`	Maximum bounds on state values, specified as a vector of length `Nx`. By default, the maximum bound for each state is `Inf`.
	`dxMin`	Minimum bounds on state derivatives that are not at steady-state, specified as a vector of length `Nx`. By default, the minimum bound for each state derivative is `-Inf`. When you specify a derivative bound, you must also set `SteadyState` to `0`.
	`dxMax`	Maximum bounds on state derivatives that are not at steady-state, specified as a vector of length `Nx`. By default, the maximum bound for each state derivative is `Inf`. When you specify a derivative bound, you must also set `SteadyState` to `0`.

Property	Description	
	Field	**Description**
	`Description`	Block state description, specified as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Inputs</td><td>Input level specifications at the operating point, returned as a vector of input specification objects. Each entry in Inputs represents the input levels of one root-level inport block in the model.

Each input specification object has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Block</td><td>Inport block name</td></tr>
<tr><td>PortWidth</td><td>Number of inport block signals</td></tr>
<tr><td>PortDimensions</td><td>Dimension of signals accepted by the inport</td></tr>
<tr><td>u</td><td>Inport block input levels at the operating point, returned as a vector of length PortWidth.

If the corresponding flag in Known field of Inputs is 1, u contains the known input values. Otherwise, u contains initial guesses for the input values.</td></tr>
<tr><td>Known</td><td>Flags indicating whether input levels are known during trimming, specified as a logical vector of length PortWidth.

• 1 — Known input level that is fixed during operating point search
• 0 (default) — Unknown input level found by optimization

To fix an input level during an operating point search, set the corresponding Known flag to 1, and specify the input value using the u property of Inputs.</td></tr>
<tr><td>Min</td><td>Minimum bounds on input levels, specified as a vector of length PortWidth. By default, the minimum bound for each input is -Inf.</td></tr>
<tr><td>Max</td><td>Maximum bounds on input levels, specified as a vector of length PortWidth. By default, the maximum bound for each input is Inf.</td></tr>
<tr><td>Description</td><td>Inport block input description, specified as a character vector.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Outputs</td><td>Output level specifications at the operating point, returned as a vector of output specification objects. Each entry in Outputs represents the output levels of one root-level outport block of the model or one trim output constraint in the model.

You can specify additional trim output constraints using addoutputspec.

Each output specification object has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Block</td><td>Outport block name</td></tr>
<tr><td>PortWidth</td><td>Number of outport block signals</td></tr>
<tr><td>PortNumber</td><td>Number of this outport in the model</td></tr>
<tr><td>y</td><td>Outport block output levels at the operating point, specified as a vector of length PortWidth.

If the corresponding flag in Known field of Outputs is 1, y contains the known output values. Otherwise, y contains initial guesses for the output values.</td></tr>
<tr><td>Known</td><td>Flags indicating whether output levels are known during trimming, specified as a logical vector of length PortWidth.

• 1 — Known output level that is fixed during operating point search
• 0 (default) — Unknown output level found by optimization

To fix an output level during an operating point search, set the corresponding Known flag to 1, and specify the output value using the y property of Outputs.</td></tr>
<tr><td>Min</td><td>Minimum bounds on output levels, specified as a vector of length PortWidth. By default, the minimum bound for each output is -Inf.</td></tr>
<tr><td>Max</td><td>Maximum bounds the output levels, specified as a vector of length PortWidth. By default, the maximum bound for each output is Inf.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td></td><td><table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Description</code></td><td>Outport block input description, specified as a character vector.</td></tr>
</table></td></tr>
<tr><td><code>Time</code></td><td>Times at which the time-varying functions in the model are evaluated, returned as a vector.</td></tr>
<tr><td><code>CustomObjFcn</code></td><td>Function providing an additional custom objective function for trimming, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder.

You can specify a custom objective function as an algebraic combination of model states, inputs, and outputs. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.</td></tr>
<tr><td><code>CustomConstrFcn</code></td><td>Function providing additional custom constraints for trimming, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder.

You can specify custom equality and inequality constraints as algebraic combinations of model states, inputs, and outputs. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.</td></tr>
<tr><td><code>CustomMappingFcn</code></td><td>Function that maps model states, inputs, and outputs to the vectors accepted by <code>CustomConstrFcn</code> and <code>CustomObjFcn</code>, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder.

For complex models, you can pass subsets of the model inputs, outputs, and states to the custom constraint and objective functions using a custom mapping function. If you specify a custom mapping, you must use the mapping for both the custom constraint function and the custom objective function. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-50.</td></tr>
</table>

Tips

- To display the operating point specification object properties, use `get`.

See Also

`addoutputspec` | `findop` | `update`

Introduced before R2006a

set

Set properties of linearization I/Os and operating points

Syntax

```
set(ob)
set(ob,'PropertyName',val)
```

Description

`set(ob)` displays all editable properties of the object, `ob`, which can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`set(ob,'PropertyName',val)` sets the property, `PropertyName`, of the object, `ob`, to the value, `val`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`ob.PropertyName = val` is an alternative notation for assigning the value, `val`, to the property, `PropertyName`, of the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

Examples

Create an operating point object for the Simulink model, `magball`:

```
op_cond=operpoint('magball');
```

Use the `set` function to get a list of all editable properties of this object:

```
set(op_cond)
```

This function returns the properties of `op_cond`.

```
ans =
     Model: {}
    States: {}
    Inputs: {}
      Time: {}
```

To set the value of a particular property of `op_cond`, provide the property name and the desired value of this property as arguments to `set`. For example, to change the name of the model associated with the operating point object from `'magball'` to `'Magnetic Ball'`, type:

```
set(op_cond,'Model','Magnetic Ball')
```

To view the property value and verify that the change was made, type:

```
op_cond.Model
```

which returns

```
ans =
Magnetic Ball
```

Because `op_cond` is a structure, you can set any properties or fields using dot-notation. First, produce a list of properties of the second `States` object within `op_cond`, as follows:

```
set(op_cond.States(2))
```

which returns

```
ans =

                    Nx: {}
                 Block: {}
             StateName: {}
                     x: {}
                    Ts: {}
            SampleType: {}
    inReferencedModel: {}
           Description: {}
```

Now, use dot-notation to set the `x` property to `8`:

```
op_cond.States(2).x=8;
```

To view the property and verify that the change was made, type

```
op_cond.States(2)
```

which displays

```
(1.) magball/Magnetic Ball Plant/Current
      x: 8
```

See Also

`findop` | `get` | `linio` | `operpoint` | `operspec` | `setlinio`

Introduced before R2006a

setlinio

Save linear analysis points to Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
setlinio(mdl,io)
setlinio(blockpath,io)
oldio = setlinio(___)
```

Description

`setlinio(mdl,io)` writes the analysis points specified in `io` to the Simulink model `mdl`.

`setlinio(blockpath,io)` sets the specified analysis points to the specified Linear Analysis Plots block or Model Verification block.

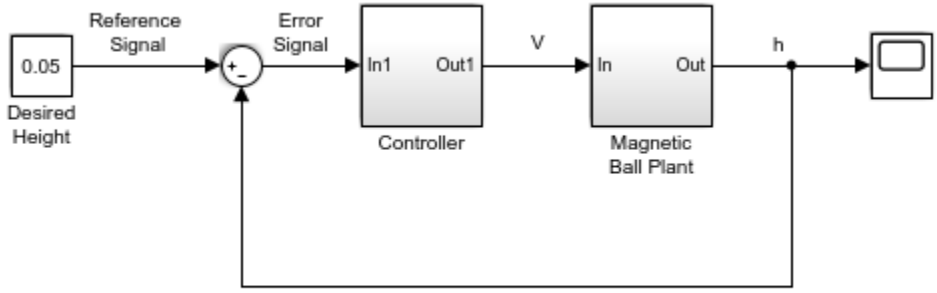
`oldio = setlinio(___)` returns the current set of analysis points in the model or block and replaces them with `io` using any of the previous syntaxes.

Examples

Set Analysis Points in Simulink Model

Open Simulink model.

```
model = 'magball';
open_system(model)
```



Create a vector of analysis points for linearizing the plant model:

- Input perturbation at the output of the Controller block
- Open-loop output at the output of the Magnetic Ball Plant block

```
io(1) = linio('magball/Controller',1,'input');
io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
```

Write the analysis points to the `magball` model.

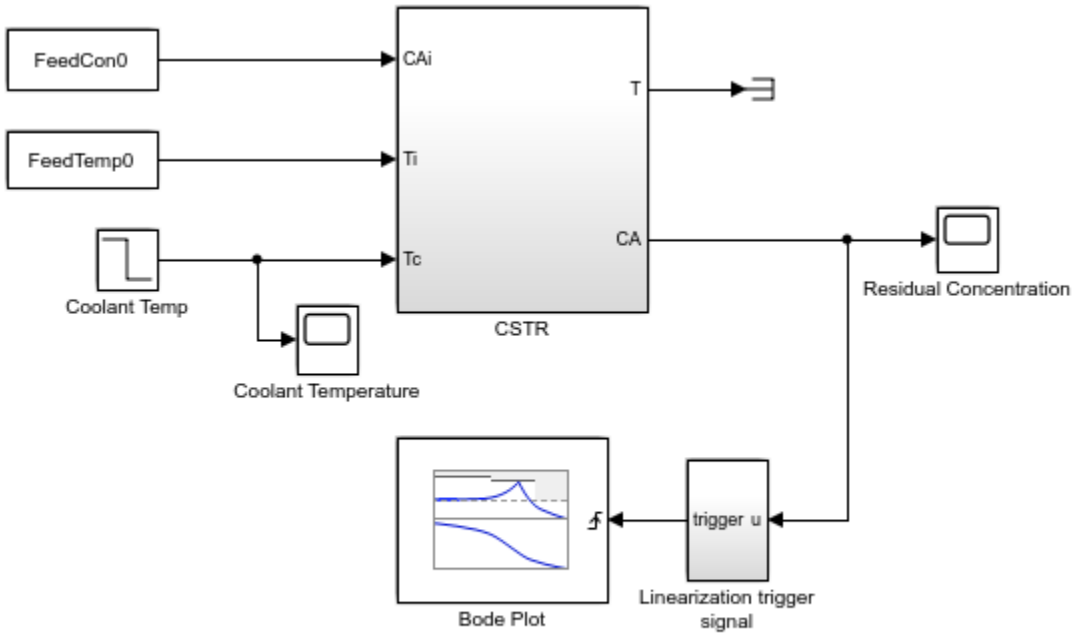
```
setlinio(model,io);
```

The analysis points in `io` are added to the model as annotations. You can then save the model to store the analysis points with the model.

Set Analysis Points in Linear Analysis Plots Block

Open Simulink model.

```
open_system('scdcstr')
```



Create analysis points for finding the transfer function between the coolant temperature and the residual concentration.

- Input perturbation at the output of the Coolant Temp block
- Output measurement at the CA output of the CSTR block

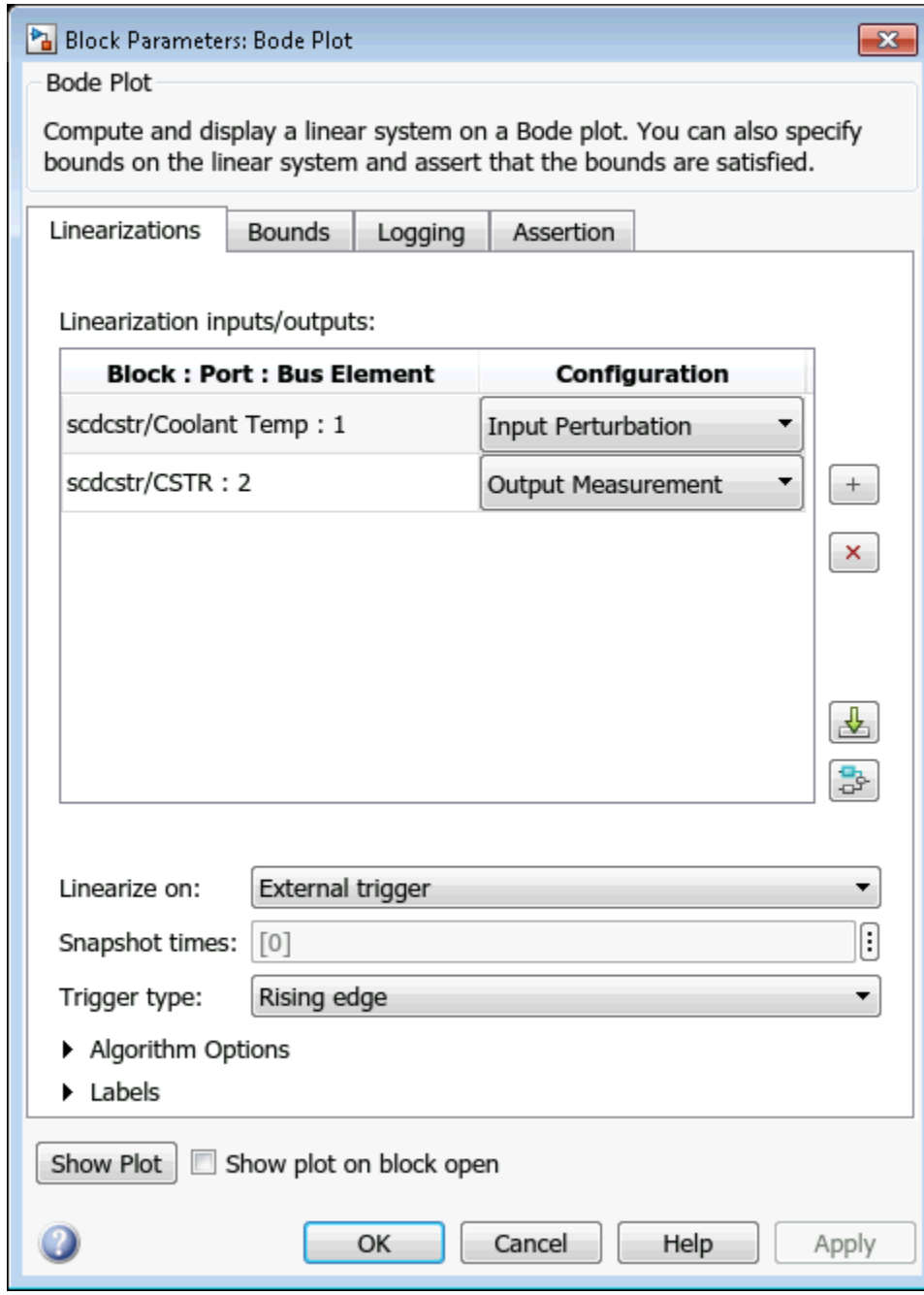
```
io(1) = linio('scdcstr/Coolant Temp',1,'input');
io(2) = linio('scdcstr/CSTR',2,'output');
```

Set the analysis points in the Bode Plot block.

```
setlinio('scdcstr/Bode Plot',io);
```

View the analysis points in the Bode Plot Block Parameters dialog box.

```
open_system('scdcstr/Bode Plot')
```

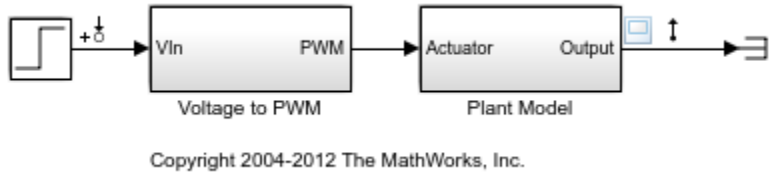

Block Parameters: Bode Plot
Bode Plot
Compute and display a linear system on a Bode plot. You can also specify bounds on the linear system and assert that the bounds are satisfied.
Linearizations
Bounds
Logging
Assertion
Linearization inputs/outputs:
Block : Port : Bus Element
Configuration
scdcstr/Coolant Temp : 1
Input Perturbation
scdcstr/CSTR : 2
Output Measurement
Linearize on:
External trigger
Snapshot times:
[0]
Trigger type:
Rising edge
Algorithm Options
Labels
Show Plot
Show plot on block open
OK
Cancel
Help
Apply

During simulation, the software linearizes the model using the specified analysis, and plots the magnitude and phase responses for the resulting linear system.

Save Old Analysis Points When Storing New Analysis Points

Open Simulink model.

```
mdl = 'scdpwm';
open_system(mdl)
```



This model is configured with analysis points for finding the combined transfer function of the PWM and plant blocks.

Create analysis points for finding the transfer function of just the plant model.

```
io(1) = linio('scdpwm/Voltage to PWM',1,'input');
io(2) = linio('scdpwm/Plant Model',1,'output');
```

Store the analysis points to the model, and save the previous analysis point configuration.

```
oldio = setlinio(mdl,io)
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdpwm/Step
- Port: 1
2. Linearization output measurement located at the following signal:
```

```
- Block: scdpwm/Plant Model
- Port: 1
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

If the model is not open or loaded into memory, `setlinio` loads the model into memory.

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set, specified as a linearization I/O object or a vector of linearization I/O objects.

Each linearization I/O object has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.

Property	Description
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-13.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

`blockpath` — Linear Analysis Plots block or Model Verification block
character vector | string

Linear Analysis Plots block or Model Verification block, specified as a character vector or string that contains its full block path. The model that contains the block must be in the current working folder or on the MATLAB path.

For more information on:

- Linear analysis plot blocks, see “Visualization During Simulation”.
- Model verification blocks, see “Model Verification”.

Output Arguments

`oldio` — Old analysis point set
linearization I/O object | vector of linearization I/O objects

Old analysis point set, returned as a linearization I/O object or a vector of linearization I/O objects.

Alternative Functionality

Simulink Model

You can also specify analysis points directly in a Simulink model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-21.

See Also

`getlinio` | `linearize` | `linio` | `slLinearizer`

Topics

“Specify Portion of Model to Linearize” on page 2-13

Introduced before R2006a

setxu

Set states and inputs in operating points

Syntax

```
op_new=setxu(op_point,x,u)
```

Description

`op_new=setxu(op_point,x,u)` sets the states and inputs in the operating point, `op_point`, with the values in `x` and `u`. A new operating point containing these values, `op_new`, is returned. The variable `x` can be a vector or a structure with the same format as those returned from a Simulink simulation. The variable `u` can be a vector. Both `x` and `u` can be extracted from another operating point object with the `getxu` function.

Examples

Initialize Operating Point Object Using State Values from Simulation

Export state values from a simulation and use the exported values to initialize an operating point object.

Open the Simulink model. This example uses the model `scdplane`.

```
open_system('scdplane')
```

Select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, select **Data Import/Export**. Check **Final states** and click **OK**. These selections save the final states of the model to the workspace after a simulation.

Simulate the model. After the simulation, a new variable, `xFinal`, appears in the workspace. This variable is a vector containing the final state values.

Create an operating point object for `scdplane`.

```
op_point = operpoint('scdplane')

 Operating Point for the Model scdplane.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdplane/Actuator Model
      x: 0
(2.) scdplane/Aircraft Dynamics Model/Transfer Fcn.1
      x: 0
(3.) scdplane/Aircraft Dynamics Model/Transfer Fcn.2
      x: 0
(4.) scdplane/Controller/Alpha-sensor Low-pass Filter
      x: 0
(5.) scdplane/Controller/Pitch Rate Lead Filter
      x: 0
(6.) scdplane/Controller/Proportional plus integral compensator
      x: 0
(7.) scdplane/Controller/Stick Prefilter
      x: 0
(8.) scdplane/Dryden Wind Gust Models/Q-gust model
      x: 0
(9.) scdplane/Dryden Wind Gust Models/W-gust model
      x: 0
      x: 0

Inputs:
----------
(1.) scdplane/u
      u: 0
```

All states are initially set to 0.

Initialize the states in the operating point object to the values in `xFinal`. Set the input to be `9`.

```
newop = setxu(op_point,xFinal,9)

 Operating Point for the Model scdplane.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdplane/Actuator Model
```

```
      x: -0.032
(2.) scdplane/Aircraft Dynamics Model/Transfer Fcn.1
      x: 0.56
(3.) scdplane/Aircraft Dynamics Model/Transfer Fcn.2
      x: 678
(4.) scdplane/Controller/Alpha-sensor Low-pass Filter
      x: 0.392
(5.) scdplane/Controller/Pitch Rate Lead Filter
      x: 0.133
(6.) scdplane/Controller/Proportional plus integral compensator
      x: 0.166
(7.) scdplane/Controller/Stick Prefilter
      x: 0.1
(8.) scdplane/Dryden Wind Gust Models/Q-gust model
      x: 0.114
(9.) scdplane/Dryden Wind Gust Models/W-gust model
      x: 0.46
      x: -2.05

Inputs:
----------
(1.) scdplane/u
      u: 9
```

Alternatives

As an alternative to the `setxu` function, set states and inputs of operating points with the Simulink Control Design GUI.

See Also

`getxu` | `initopspec` | `operpoint` | `operspec`

Introduced before R2006a

slLinearizer

Interface for batch linearization of Simulink models

Syntax

```
sllin = slLinearizer(mdl)
sllin = slLinearizer(mdl,pt)
sllin = slLinearizer(mdl,param)
sllin = slLinearizer(mdl,op)
sllin = slLinearizer(mdl,blocksub)
sllin = slLinearizer(mdl,options)
sllin = slLinearizer(mdl,pt,op,param,blocksub,options)
```

Description

`sllin = slLinearizer(mdl)` creates an `slLinearizer` interface, `sllin`, for linearizing the Simulink model, `mdl`. The interface adds the linear analysis points marked in the model as analysis points on page 13-254 of `sllin`. The interface also adds the linear analysis points that imply an opening as permanent openings on page 13-255.

`sllin = slLinearizer(mdl,pt)` adds the specified point to the list of analysis points for `sllin`, ignoring linear analysis points marked in the model.

`sllin = slLinearizer(mdl,param)` specifies the parameters whose values you want to vary when linearizing the model.

`sllin = slLinearizer(mdl,op)` specifies the operating points for linearizing the model.

`sllin = slLinearizer(mdl,blocksub)` specifies substitute linearizations of blocks and subsystems. Use this syntax, for example, to specify a custom linearization for a block. You can also use this syntax for blocks that do not linearize successfully, such as blocks with discontinuities or triggered subsystems.

`sllin = slLinearizer(mdl,options)` configures the linearization algorithm options.

`sllin = slLinearizer(mdl,pt,op,param,blocksub,options)` uses any combination of the input arguments `pt`, `op`, `param`, `blocksub`, and `options` to create `sllin`.

For example, use any of the following:

- `sllin = slLinearizer(mdl,pt,param)`
- `sllin = slLinearizer(mdl,op,param)`.

If you do not specify `pt`, the interface adds the linear analysis points marked in the model as analysis points. The interface also adds linear analysis points that imply an opening as permanent openings.

Object Description

`slLinearizer` provides an interface between a Simulink model and the linearization commands `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use `slLinearizer` to efficiently batch linearize a model. You can configure the `slLinearizer` interface to linearize a model at a range of operating points and specify variations for model parameter values. Use interface analysis points on page 13-254 and permanent openings on page 13-255 to obtain linearizations for any open-loop or closed-loop transfer function from a model. Analyze the stability, or time-domain or frequency-domain characteristics of the linearized models.

If you changed any interface properties since the last linearization, commands that extract linearizations from the `slLinearizer` interface recompile the Simulink model. If you made calls to specific functions since the last linearization, the commands also recompile the Simulink model. These functions include `addPoint`, `addOpening`, `removePoint`, `removeOpening`, `removeAllPoints`, and `removeAllOpenings`.

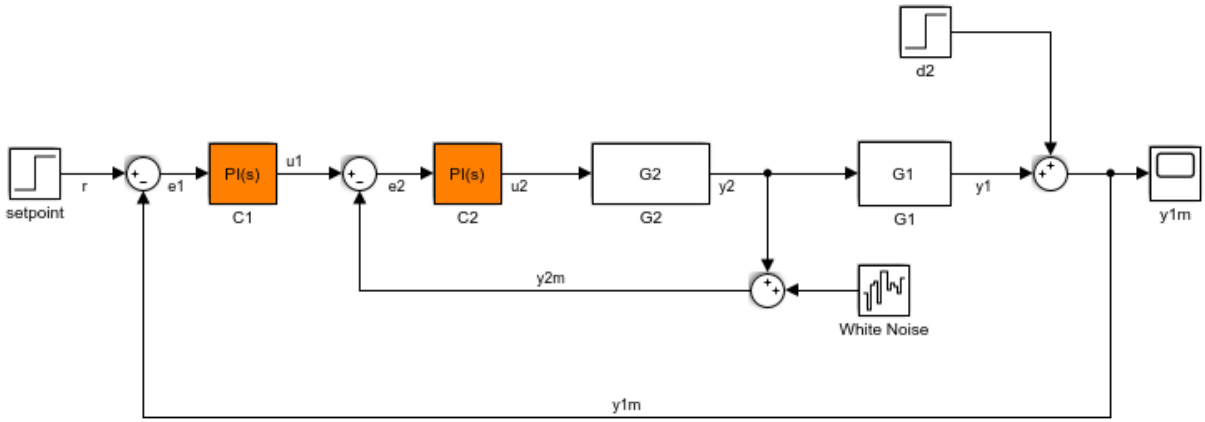
Examples

Create and Configure slLinearizer Interface for Batch Linear Analysis

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points to the interface to extract open- or closed-loop transfer functions from the model. Configure the interface to vary parameters and operating points.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model. Add the signals `r`, `u1`, `u2`, `y1`, `y2`, `y1m`, and `y2m` to the interface.

```
sllin = slLinearizer(mdl,{'r','u1','u2','y1','y2','y1m','y2m'});
```

`scdcascade` contains two PID Controller blocks, `C1` and `C2`. Suppose you want to vary the proportional and integral gains of `C2`, `Kp2` and `Ki2`, in the 10% range. Create a structure to specify the parameter variations.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid]=ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;
```

`params` specifies a 3x5 parameter grid. Each point in this grid corresponds to a combination of the `Kp2` and `Ki2` parameter values.

Specify `params` as the `Parameters` property of `sllin`.

```
sllin.Parameters = params;
```

Now, when you use commands such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`, the software returns a linearization for each parameter grid point specified by `sllin.Parameters`.

Suppose you want to linearize the model at multiple snapshot times, for example at `t = {0,1,2}`. To do so, configure the `OperatingPoints` property of `sllin`.

```
sllin.OperatingPoints = [0 1 2];
```

You can optionally configure the linearization options and specify substitute linearizations for blocks and subsystems in your model. After fully configuring `sllin`, use the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` commands to linearize the model as required.

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string.

Example: `'scdcascade'`

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point on page 13-254 to be added to the list of analysis points for `sllin`, specified as:

- Character vector or string — Analysis point identifier that can be any of the following:
  - Signal name, for example `pt = 'torque'`
  - Block path for a block with a single output port, for example `pt = 'Motor/PID'`
  - Block path and port originating the signal, for example `pt = 'Engine Model/1'`

- Cell array of character vectors or string array — Specifies multiple analysis point identifiers. For example:

  ```
  pt = {'torque','Motor/PID','Engine Model/1'}
  ```

- Vector of linearization I/O objects — Create `pt` using `linio`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1,'input');
  pt(2) = linio('scdcascade/Sum',1,'output');
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output.

  The interface adds all the points specified by `pt` and ignores their I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings on page 13-255.

`param` — Parameter samples

structure | structure array

Parameter samples for linearizing `mdl`, specified as:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  ```

```
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

For more information, see “Specify Parameter Samples for Batch Linearization” on page 3-62.

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `sllin.OperatingPoints` with operating point objects only, the software uses single model compilation.

For an example showing how batch linearization with parameter sampling works, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32.

To compute the offsets required by the LPV System block, specify `param`, and set `sllin.Options.StoreOffsets` to `true`. You can then return additional linearization information when calling linearization functions such as `getIOTransfer`, and extract the offsets using `getOffsetsForLPV`.

`op` — Operating point for linearizing `mdl`

operating point object | array of operating point objects | vector of positive scalars

Operating point for linearizing `mdl`, specified as:

- Operating point object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of operating point objects, specifying multiple operating points.

  To create an array of operating point objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-65.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `sllin.Parameters`, then specify `op` as one of the following:

- Single operating point.
- Array of operating point objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see "Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations" on page 3-25.
- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`blocksub` — Substitute linearizations for blocks and model subsystems

structure | structure array

Substitute linearizations for blocks and model subsystems, specified as a structure or an n-by-1 structure array, where n is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

You can batch linearize your model by specifying multiple substitute linearizations for a block. Use this functionality, for example, to study the effects of varying the linearization of a Saturation block on the model dynamics.

Each substitute linearization structure has the following fields:

`Name` — Block path

Block path of the block for which you want to specify the linearization, specified as a character vector or string.

`Value` — Substitute linearization

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.
- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block

specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.

- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:
  - Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
  - Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for more than one block, the dimensions of the arrays must match.
- Structure with the following fields:

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following • MATLAB expression • Name of a "Custom Linearization Function" on page 13-255 in your current working directory or on the MATLAB path. The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`

Field	Description
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.
`ParameterValues`	Linearization function parameter values, specified as an vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`options` — Linearization algorithm options
`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` option set.

Properties

`slLinearizer` object properties include:

`Parameters`

Parameter samples for linearizing `mdl`, specified as a structure or a structure array.

Set this property using the `param` input argument or dot notation (`sllin.Parameters = param`). `param` must be one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:
  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar

variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

```
param.Name = 'V(1)';
```

- `Value` — Parameter sample values, specified as a double array.

For example, vary the value of parameter `A` in the 10% range:

```
param.Name = 'A';
param.Value = linspace(0.9*A,1.1*A,3);
```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,3));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `sllin.OperatingPoints` with operating point objects only, the software uses single model compilation.

`OperatingPoints`

Operating points for linearizing `mdl`, specified as an operating point object, array of operating point objects, or array of positive scalars.

Set this property using the `op` input argument or dot notation (`sllin.OperatingPoints = op`). `op` must be one of the following:

- Operating point object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of operating point objects, specifying multiple operating points.

  To create an array of operating point objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61.

  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-65.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `sllin.Parameters`, then specify `op` as one of the following:

- Single operating point.
- Array of operating point objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-25.
- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`BlockSubstitutions`

Substitute linearizations for blocks and model subsystems, specified as a structure or structure array.

Use this property to specify a custom linearization for a block or subsystem. You also can use this syntax for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

Set this property using the `blocksub` input argument or dot notation (`sllin.BlockSubstitutions = blocksubs`). For information about the required structure, see `blocksub`.

`Options`

Linearization algorithm options, specified as an option set created using `linearizeOptions`.

Set this property using the `options` input argument or dot notation (`sllin.Options = options`).

`Model`

Name of the Simulink model to be linearized, specified as a character vector by the input argument `mdl`.

`TimeUnit`

Unit of the time variable. This property specifies the time units for linearized models returned by `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use any of the following values:

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

Default: `'seconds'`

Object Functions

addPoint	Add signal to list of analysis points for slLinearizer or slTuner interface
addOpening	Add signal to list of openings for slLinearizer or slTuner interface
getPoints	Get list of analysis points for slLinearizer or slTuner interface
getOpenings	Get list of openings for slLinearizer or slTuner interface
getIOTransfer	Transfer function for specified I/O set using slLinearizer or slTuner interface
getLoopTransfer	Open-loop transfer function at specified point using slLinearizer or slTuner interface
getSensitivity	Sensitivity function at specified point using slLinearizer or slTuner interface

getCompSensitivity	Complementary sensitivity function at specified point using slLinearizer or slTuner interface
removePoint	Remove point from list of analysis points in slLinearizer or slTuner interface
removeAllPoints	Remove all points from list of analysis points in slLinearizer or slTuner interface
removeAllOpenings	Remove all openings from list of permanent openings in slLinearizer or slTuner interface
refresh	Resynchronize slLinearizer or slTuner interface with current model state

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

Field	Description
`BlockName`	Name of the block for which you are specifying a custom linearization.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Parameters</td><td>Block parameter values, specified as a structure array with Name and Value fields. Parameters contains the names and values of the parameters you specify in the blocksub.Value.ParameterNames and blocksub.Value.ParameterValues fields.</td></tr>
<tr><td>Inputs</td><td>Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in Inputs has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>BlockName</td><td>Full block path of the block whose output connects to the corresponding block input.</td></tr>
<tr><td>PortIndex</td><td>Output port of the block specified by BlockName that connects to the corresponding block input.</td></tr>
<tr><td>Values</td><td>Value of the signal specified by BlockName and PortIndex. If this signal is a vector signal, then Values is a vector with the same dimension.</td></tr>
</table></td></tr>
<tr><td>ny</td><td>Number of output channels of the block linearization.</td></tr>
<tr><td>nu</td><td>Number of input channels of the block linearization.</td></tr>
<tr><td>BlockLinearization</td><td>Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using BlockLinearization.</td></tr>
</table>

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
```

```
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize`

Topics

“What Is Batch Linearization?” on page 3-2
“How the Software Treats Loop Openings” on page 2-39
“Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10
“Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61
“Specify Parameter Samples for Batch Linearization” on page 3-62
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-41
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32

Introduced in R2013b

addOpening

Add signal to list of openings for `slLinearizer` or `slTuner` interface

Syntax

```
addOpening(s,pt)

addOpening(s,blk,port_num)
addOpening(s,blk,port_num,bus_elem_name)
```

Description

`addOpening(s,pt)` adds the specified point (signal) to the list of permanent openings on page 13-264 for the `slLinearizer` or `slTuner` interface, `s`.

Use permanent openings to isolate a specific model component for the purposes of linearization and tuning. Suppose you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

`addOpening(s,blk,port_num)` adds the signal at the specified output port of the specified block as a permanent opening for `s`.

`addOpening(s,blk,port_num,bus_elem_name)` adds the specified bus element as a permanent opening.

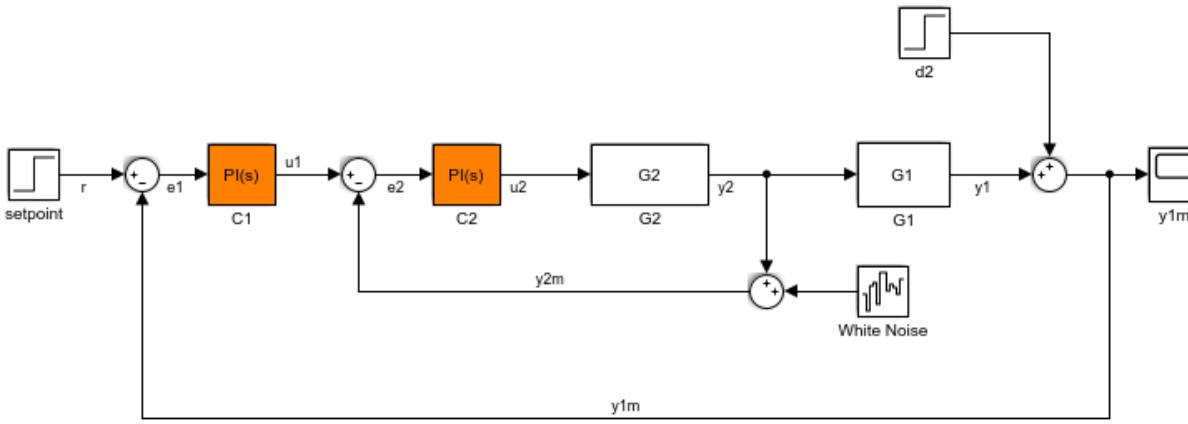
Examples

Add Opening Using Signal Name

Suppose you want to analyze only the inner-loop dynamics of the `scdcascade` model. Add the outer-loop feedback signal, `y1m`, as a permanent opening of an `slLinearizer` interface.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add the `y1m` signal as a permanent opening of `sllin`.

```
addOpening(sllin,'y1m');
```

View the currently defined analysis points within `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
1 Permanent openings:
-------------------------
Opening 1:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m
```

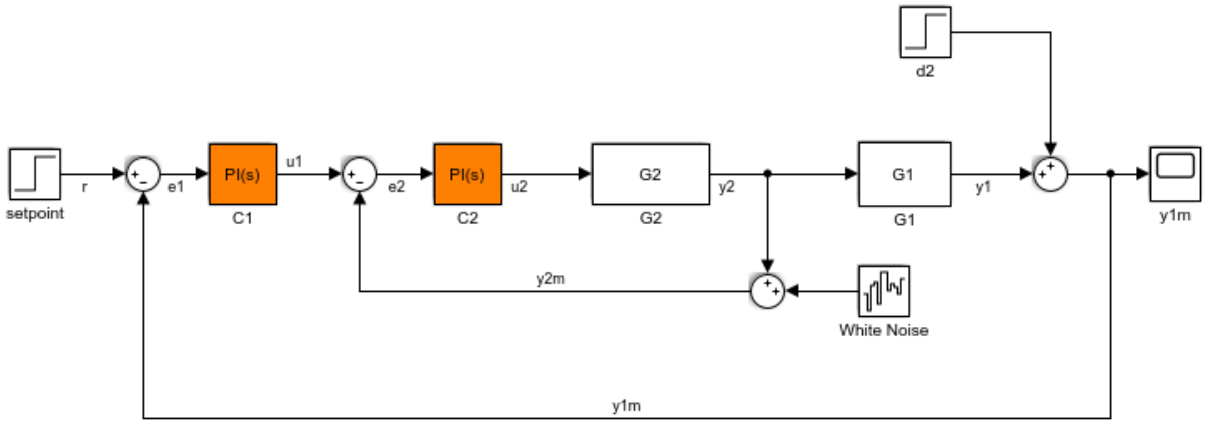
```
Properties with dot notation get/set access:
      Parameters          : []
      OperatingPoints     : [] (model initial condition will be used.)
      BlockSubstitutions  : []
      Options             : [1x1 linearize.LinearizeOptions]
```

Add Opening Using Block Path and Port Number

Suppose you want to analyze only the inner-loop dynamics of the `scdcascade` model. Add the outer-loop feedback signal, `y1m`, as a permanent opening of an `slLinearizer` interface.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add the `y1m` signal as a permanent opening of `sllin`.

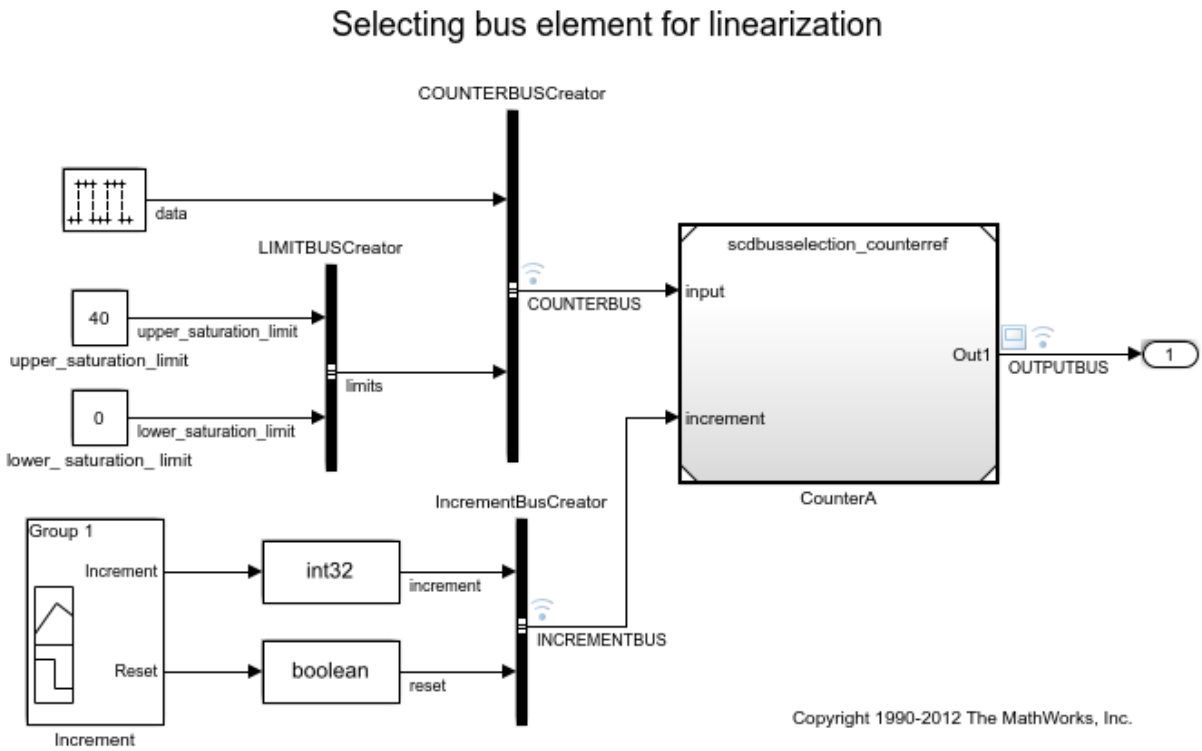
```
addOpening(sllin,'scdcascade/Sum',1);
```

The `y1m` signal originates at the first (and only) port of the `scdcascade/Sum` block.

Add Bus Elements as Openings

Open the `scdbusselection` model.

```
mdl = 'scdbusselection';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

The `COUNTERBUS` signal of `scdbusselection` contains multiple bus elements. Add the `upper_saturation_limit` and `data` bus elements as openings to `sllin`. When adding

elements within a nested bus structure, use dot notation to access the elements of the nested bus, for example `limits.upper_saturation_limit`.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUSCreator'};
port_num = [1 1];
bus_elem_name = {'limits.upper_saturation_limit','data'};
```

Both bus elements originate at the first (and only) port of the `scdbusselection/COUNTERBUSCreator` block. Therefore, `blk` and `port_num` repeat the same element twice.

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Opening

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Opening to add to the list of permanent openings on page 13-264 for `s`, specified as:

- Character vector or string — Signal identifier that can be any of the following:
  - Signal name, for example `'torque'`
  - Block path for a block with a single output port, for example `'Motor/PID'`
  - Path to block and port originating the signal, for example `'Engine Model/1'` or `'Engine Model/torque'`
- Cell array of character vectors or string array — Specifies multiple signal identifiers. For example, `pt = {'Motor/PID','Engine Model/1'}`.
- Vector of linearization I/O objects — Use `linio` to create `pt`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1)
  pt(2) = linio('scdcascade/Sum',1,'output')
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output. However, the software ignores the I/O types and adds them both to the list of permanent openings for `s`.

`blk` — Block path identifying block where opening originates

character vector (default) | string | cell array of character vectors | string array

Block path identifying the block where the opening originates, specified as a character vector or cell array of character vectors.

Dimensions of `blk`:

- For a single opening, specify `blk` as a character vector or string.

  For example, `blk = 'scdcascade/C1'`.
- For multiple openings, specify `blk` as a cell array of character vectors or string array. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `blk = {'scdcascade/C1','scdcascade/Sum'}`.

`port_num` — Port where opening originates

positive integer (default) | vector of positive integers

Port where the opening originates, specified as a positive integer or a vector of positive integers.

Dimensions of `port_num`:

- For a single opening, specify `port_num` as a positive integer.

  For example, `port_num = 1`.
- For multiple openings, specify `port_num` as a vector of positive integers. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `port_num = [1 1]`.

`bus_elem_name` — Bus element name

character vector (default) | string | cell array of character vectors | string array

Bus element name, specified as a character vector or cell array of character vectors.

Dimensions of `bus_elem_name`:

- For a single opening, specify `bus_elem_name` as a character vector or string.

  For example, `bus_elem_name = 'data'`.

- For multiple openings, specify `bus_elem_name` as a cell array of character vectors or string array. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `bus_elem_name = {'limits.upper_saturation_limit','data'}`.

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addBlock` | `addPoint` | `linio` | `removeAllOpenings` | `removeOpening` | `slLinearizer` | `slTuner`

Introduced in R2013b

addPoint

Add signal to list of analysis points for `slLinearizer` or `slTuner` interface

Syntax

```
addPoint(s,pt)

addPoint(s,blk,port_num)
addPoint(s,blk,port_num,bus_elem_name)
```

Description

`addPoint(s,pt)` adds the specified point to the list of analysis points on page 13-271 for the `slLinearizer` or `slTuner` interface, `s`.

Analysis points are model signals that can be used as input, output, or loop-opening locations for analysis and tuning purposes. You use analysis points as inputs to the linearization commands of `s`: `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open- or closed-loop transfer function in a model. You can also use analysis points to specify tuning goals for `systune`.

`addPoint(s,blk,port_num)` adds the point that originates at the specified output port of the specified block as an analysis point for `s`.

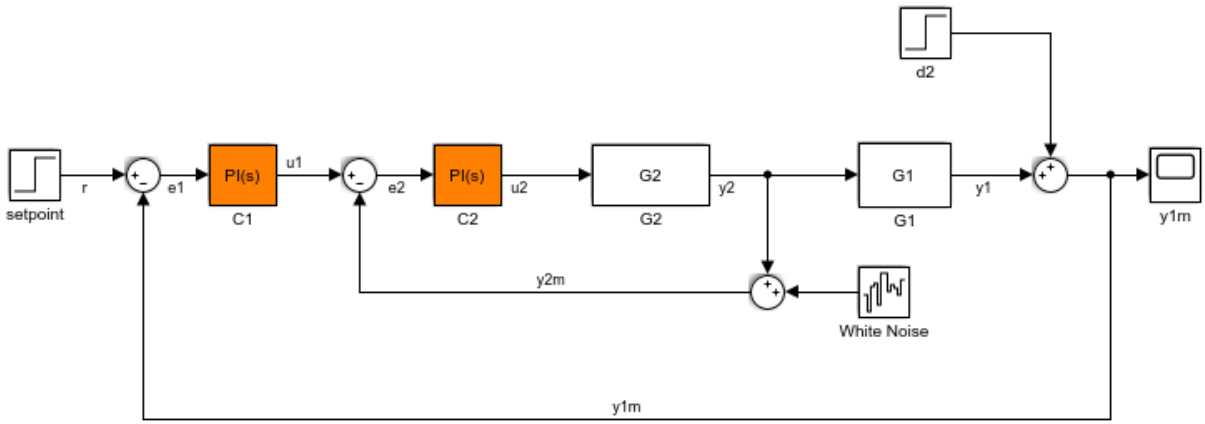
`addPoint(s,blk,port_num,bus_elem_name)` adds the specified bus element as an analysis point.

Examples

Add Analysis Point Using Signal Name

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add `u1` and `y1` as analysis points for `sllin`.

```
addPoint(sllin,{'u1','y1'});
```

View the currently defined analysis points within `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

2 Analysis points:
--------------------------
Point 1:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1
Point 2:
- Block: scdcascade/G1
- Port: 1
- Signal Name: y1
```

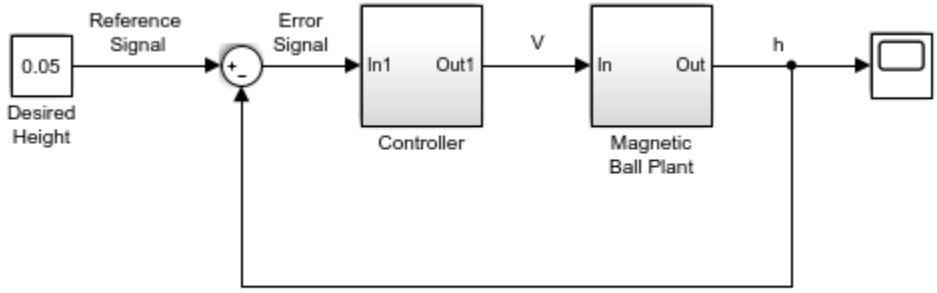
```
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters          : []
      OperatingPoints     : [] (model initial condition will be used.)
      BlockSubstitutions  : []
      Options             : [1x1 linearize.LinearizeOptions]
```

Add Analysis Points Using Block Path and Port Number

Suppose you want to linearize the magball model and obtain a transfer function from the reference input to the plant output. Add the signals originating at the `Desired Height` and `Magnetic Ball Plant` blocks as analysis points to an `slLinearizer` interface.

Open the `magball` model.

```
mdl = 'magball';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

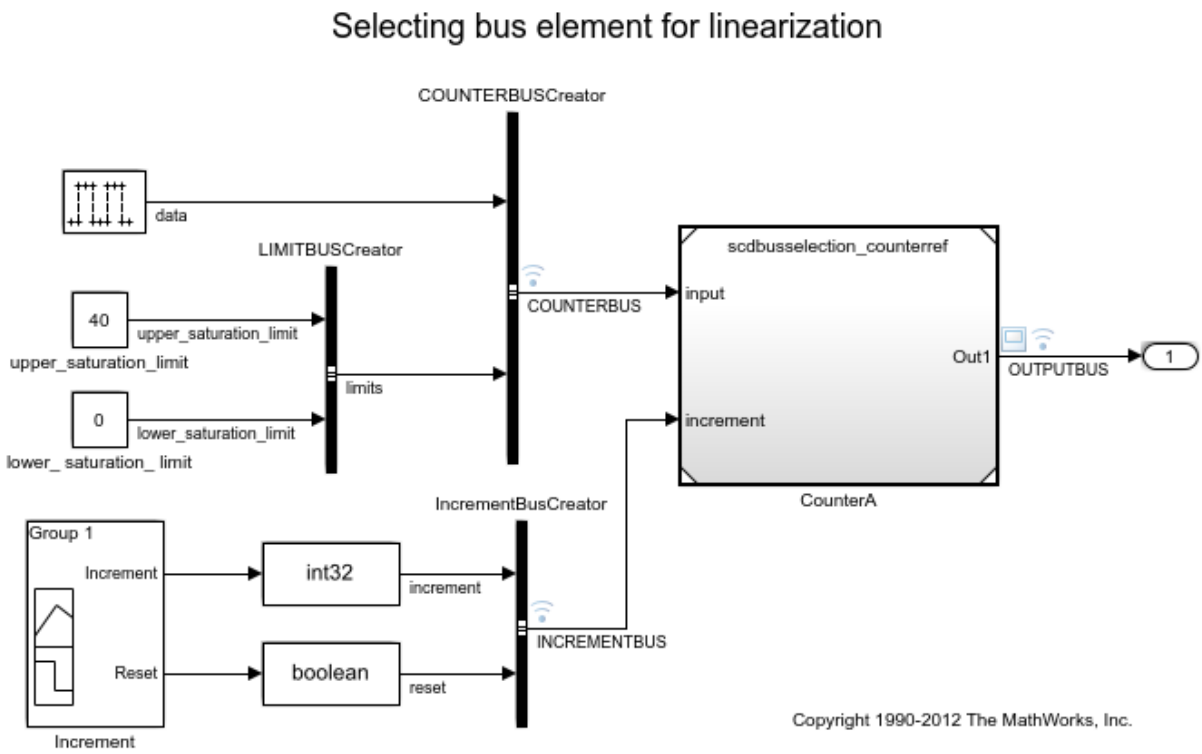
Add the signals originating at the `Design Height` and `Magnetic Ball Plant` blocks as analysis points of `sllin`. Both signals originate at the first (and only) port of the respective blocks.

```
blk = {'magball/Desired Height','magball/Magnetic Ball Plant'};
port_num = [1 1];
addPoint(sllin,blk,port_num);
```

Add Bus Elements as Analysis Points

Open the `scdbusselection` model.

```
mdl = 'scdbusselection';
open_system(mdl);
```



Create an `slLinearizer` interface model.

```
sllin = slLinearizer(mdl);
```

The COUNTERBUS signal of scdbusselection contains multiple bus elements. Add the upper_saturation_limit and data bus elements as analysis points to sllin. When adding elements within a nested bus structure, use dot notation to access the elements of the nested bus, for example limits.upper_saturation_limit.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUSCreator'};
port_num = [1 1];
bus_elem_name = {'limits.upper_saturation_limit','data'};
addPoint(sllin,blk,port_num,bus_elem_name);
```

Both bus elements originate at the first (and only) port of the scdbusselection/COUNTERBUSCreator block. Therefore, blk and port_num repeat the same element twice.

Input Arguments

s — Interface to Simulink model

slLinearizer interface | slTuner interface

Interface to a Simulink model, specified as either an slLinearizer interface or an slTuner interface.

pt — Analysis point

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point to add to the list of analysis points on page 13-271 for s, specified as:

- Character vector or string — Signal identifier that can be any of the following:
  - Signal name, for example 'torque'
  - Block path for a block with a single output port, for example 'Motor/PID'
  - Path to block and port originating the signal, for example 'Engine Model/1' or 'Engine Model/torque'
- Cell array of character vectors or string array — Specifies multiple signal identifiers.
- Vector of linearization I/O objects — Use linio to create pt. For example:

```
pt(1) = linio('scdcascade/setpoint',1)
pt(2) = linio('scdcascade/Sum',1,'output')
```

Here, `pt(1)` specifies an input, and `pt(2)` specifies an output. The interface adds all the signals specified by `pt` and ignores the I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings.

`blk` — Block path identifying block where analysis point originates

character vector (default) | string | cell array of character vectors | string array

Block path identifying the block where the analysis point originates, specified as a:

- Character vector or string to specify a single point, for example `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array to specify multiple points, for example `blk = {'scdcascade/C1','scdcascade/Sum'}`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

`port_num` — Port where analysis point originates

positive integer (default) | vector of positive integers

Port where the analysis point originates, specified as a:

- Positive integer to specify a single point, for example `port_num = 1`.
- Vector of positive integers to specify multiple points, for example `port_num = [1 1]`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

`bus_elem_name` — Bus element name

character vector (default) | string | cell array of character vectors | string array

Bus element name, specified as a:

- Character vector or string to specify a single point, for example `bus_elem_name = 'data'`.
- Cell array of character vectors or string array to specify multiple points, for example `bus_elem_name = {'limits.upper_saturation_limit','data'}`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use

permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `linio` | `removeAllPoints` | `removePoint` | `slLinearizer` | `slTuner`

Introduced in R2013b

getCompSensitivity

Complementary sensitivity function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
sys = getCompSensitivity(s,pt)
sys = getCompSensitivity(s,pt,temp_opening)
sys = getCompSensitivity(___,mdl_index)
[sys,info] = getCompSensitivity(___)
```

Description

`sys = getCompSensitivity(s,pt)` returns the complementary sensitivity function on page 13-285 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 13-287 specified for `s` when it calculates `sys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getCompSensitivity` performs multiple linearizations and returns an array of complementary sensitivity functions.

`sys = getCompSensitivity(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the complementary sensitivity function of an inner loop with the outer loop open.

`sys = getCompSensitivity(___,mdl_index)` returns a subset of the batch linearization results. `mdl_index` specifies the index of the linearizations of interest, in addition to any of the input arguments in previous syntaxes.

Use this syntax for efficient linearization, when you want to obtain the complementary sensitivity function for only a subset of the batch linearization results.

`[sys,info] = getCompSensitivity(___)` returns additional linearization information.

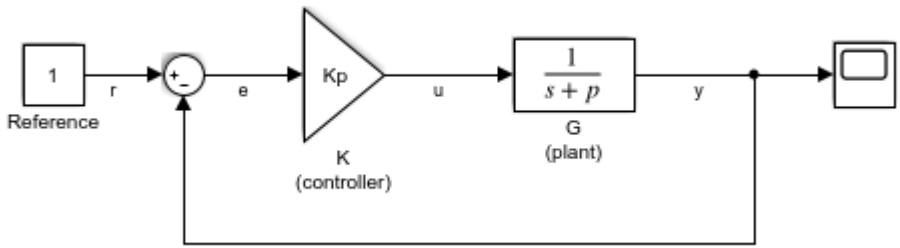
Examples

Obtain Complementary Sensitivity Function at Analysis Point

Obtain the complementary sensitivity function, calculated at the plant output, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:
$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the complementary sensitivity function at the plant output, use the `y` signal as the analysis point. Add this point to `sllin`.

```
addPoint(sllin,'y');
```

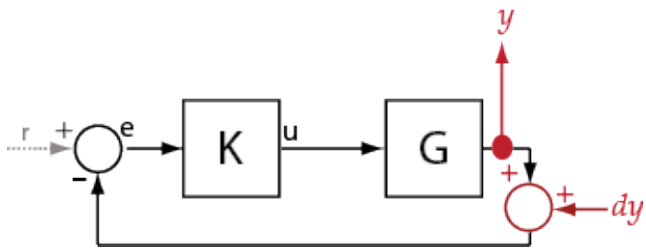
Obtain the complementary sensitivity function at `y`.

```
sys = getCompSensitivity(sllin,'y');
tf(sys)
```

```
ans =
```

```
  From input "y" to output "y":
   -3
  -----
  s + 8

Continuous-time transfer function.
```

The software adds a linearization output at `y`, followed by a linearization input, `dy`.



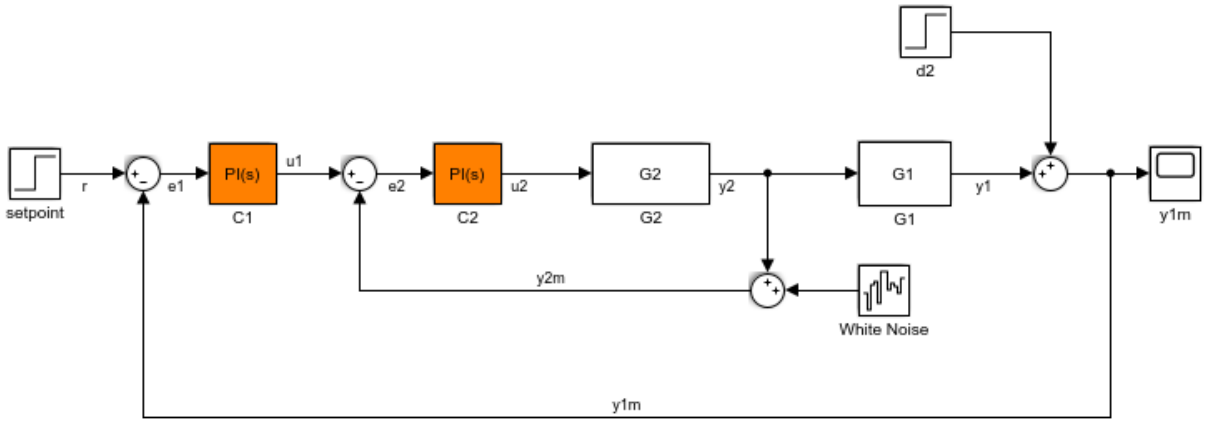
`sys` is the transfer function from `dy` to `y`, which is equal to $-(I+GK)^{-1}GK$.

Specify Temporary Loop Opening for Complementary Sensitivity Function Calculation

For the `scdcascade` model, obtain the complementary sensitivity function for the inner-loop at `y2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the complementary sensitivity transfer function for the inner loop at `y2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'});
```

Obtain the complementary sensitivity function for the inner loop at `y2`.

```
sys = getCompSensitivity(sllin,'y2','y1m');
```

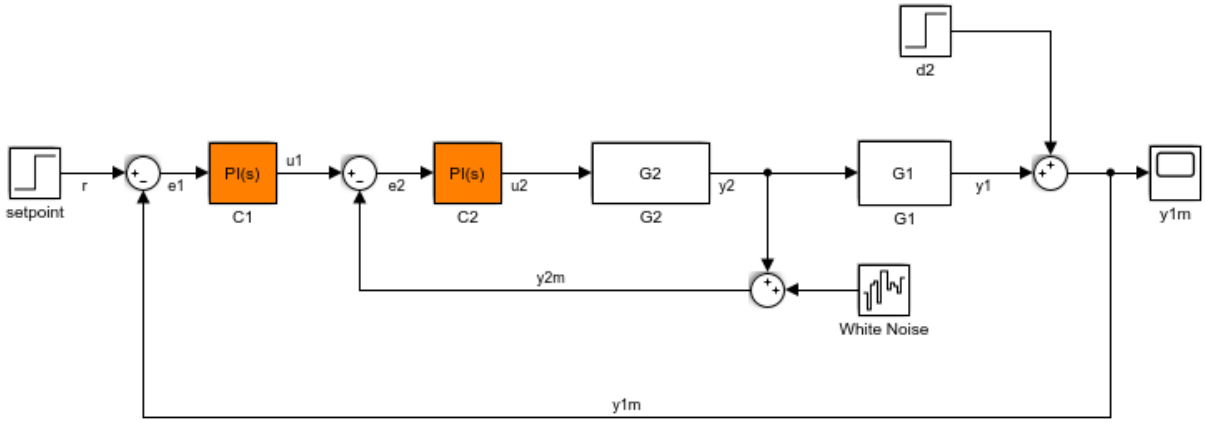
Here, `'y1m'`, the third input argument, specifies a temporary opening for the outer loop.

Obtain Complementary Sensitivity Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, calculate the complementary sensitivity function for the inner loop for the maximum value of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid]=ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the complementary sensitivity of the inner loop, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'})
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

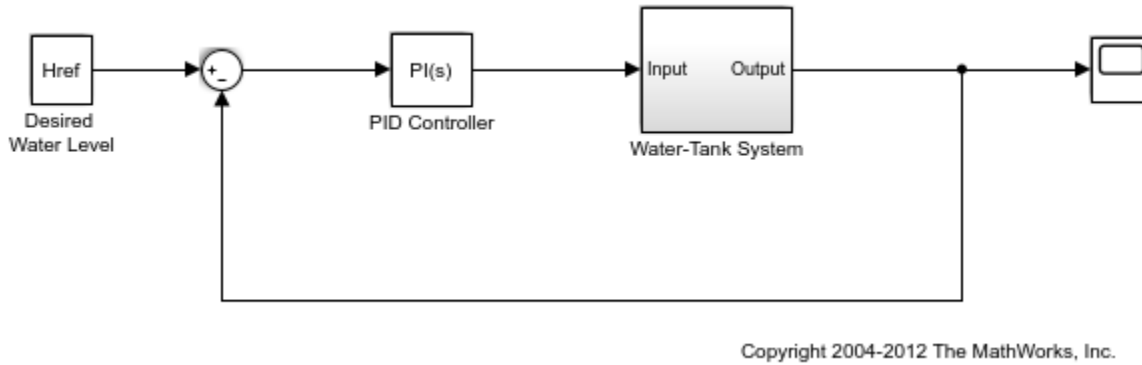
Obtain the complementary sensitivity transfer function at `y2`.

```
sys = getCompSensitivity(sllin,'y2','y1m',mdl_index);
```

Obtain Offsets from Complementary Sensitivity Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the complementary sensitivity function at `y`, and obtain the corresponding linearization offsets.

```
[sys,info] = getCompSensitivity(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [2x1 double]
           dx: [2x1 double]
            u: 1
            y: 1
    StateName: {2x1 cell}
    InputName: {'watertank/Water-Tank System'}
   OutputName: {'watertank/Water-Tank System'}
           Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 13-286 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point

does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

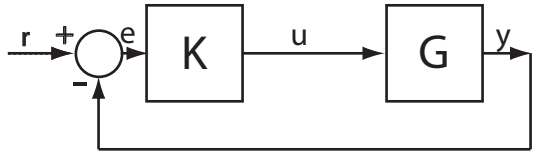
You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

For example, `pt = 'y1m'`.

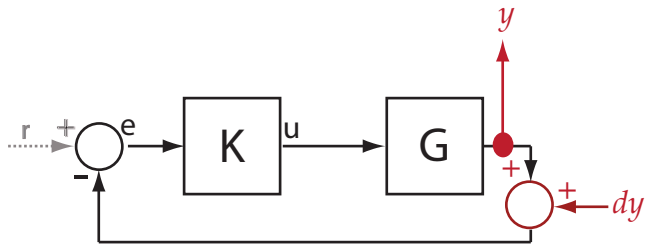
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `sys`, the software adds a linearization output, followed by a linearization input at `pt`.

Consider the following model:

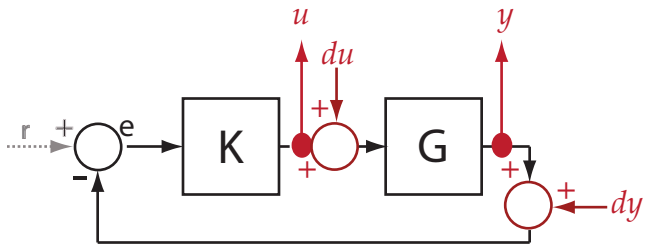


Specify `pt` as `'y'`:



The software computes `sys` as the transfer function from `dy` to `y`.

If you specify `pt` as multiple signals, for example `pt = {'u','y'}`, the software adds a linearization output, followed by a linearization input at each point.



`du` and `dy` are linearization inputs, and `u` and `y` are linearization outputs. The software computes `sys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `temp_opening = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.
- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`sys` — Complementary sensitivity function

state-space model

Complementary sensitivity function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `sys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `sys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as a state-space model array of the same size as the parameter grid.

- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as a state-space model array of size `N`-by-`p`.

`info` — Linearization information

structure

Linearization information, returned as a structure with the following field:

`Offsets` — Linearization offsets

`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `sys` is a single state-space model, then `Offsets` is a structure.
- If `sys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `sys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `sys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `sys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `sys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `sys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `sys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `sys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `sys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see "Approximating Nonlinear Behavior Using an Array of LTI Systems" on page 3-91.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `sys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `sys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `sys`.

`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results

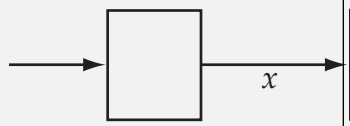
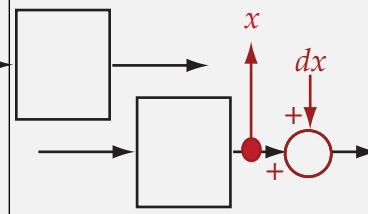
using a `LinearizationAdvisor` object, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Definitions

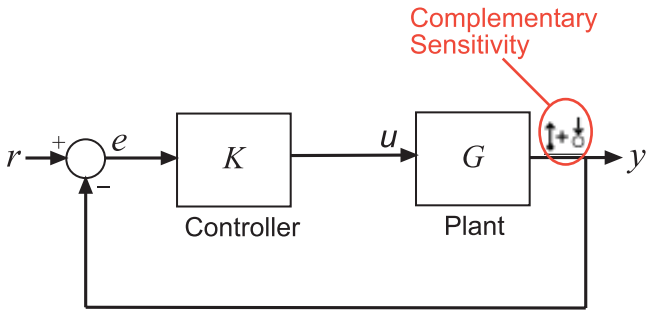
Complementary Sensitivity Function

The complementary sensitivity function at a point is the transfer function from an additive disturbance at the point to a measurement at the same point. In contrast to the sensitivity function, the disturbance is added *after* the measurement.

To compute the complementary sensitivity function at an analysis point, `x`, the software adds a linearization output at `x`, followed by a linearization input, `dx`. The complementary sensitivity function is the transfer function from `dx` to `x`.

Analysis Point in Simulink Model	How `getCompSensitivity` Interprets Analysis Point	Complementary Sensitivity Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the complementary sensitivity function at `y`:



Here, the software adds a linearization output at `y`, followed by a linearization input, `dy`. The complementary sensitivity function at `y`, T, is the transfer function from `dy` to `y`. T is calculated as follows:

$$y = -GK(y + dy)$$
$$\rightarrow y = -GKy - GKdy$$
$$\rightarrow (I + GK)y = -GKdy$$
$$\rightarrow y = \underbrace{-(I + GK)^{-1}GK}_{T}\, dy.$$

Here I is an identity matrix of the same size as GK. The complementary sensitivity transfer function at `y` is equal to `-1` times the closed-loop transfer function from `r` to `y`.

Generally, the complementary sensitivity function, T, computed from reference signals to plant outputs, is equal to $I–S$. Here S is the sensitivity function at the point, and I is the identity matrix of commensurate size. However, because `getCompSensitivity` adds the linearization output and input *at the same point*, T, as returned by `getCompSensitivity`, is equal to $S–I$.

The software does not modify the Simulink model when it computes the complementary sensitivity function.

Analysis Point

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see "Mark Signals of Interest for Control System Analysis and Design" on page 2-48 and "Mark Signals of Interest for Batch Linearization" on page 3-13.

Permanent Loop Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `addPoint` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `slLinearizer` | `slTuner`

Topics

"How the Software Treats Loop Openings" on page 2-39
"Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32
"Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41
"Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48

Introduced in R2013b

getIOTransfer

Transfer function for specified I/O set using `slLinearizer` or `slTuner` interface

Syntax

```
sys = getIOTransfer(s,in,out)
sys = getIOTransfer(s,in,out,temp_opening)

sys = getIOTransfer(s,ios)

sys = getIOTransfer(___,mdl_index)

[sys,info] = getIOTransfer(___)
```

Description

`sys = getIOTransfer(s,in,out)` returns the transfer function for the specified inputs and outputs on page 13-301 for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 13-308 specified for `s` when it calculates `sys`. For information on how `getIOTransfer` treats `in` and `out`, see "Transfer Functions" on page 13-301. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getIOTransfer` performs multiple linearizations and returns an array of transfer functions.

`sys = getIOTransfer(s,in,out,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to obtain the transfer function of the controller in series with the plant, with the feedback loop open.

`sys = getIOTransfer(s,ios)` returns the transfer function for the inputs and outputs specified by `ios` for the model associated with `s`. Use the `linio` command to create `ios`. The software enforces the linearization I/O type of each signal specified in `ios` when it calculates `sys`. The software also enforces all the permanent loop openings specified for `s`.

`sys = getIOTransfer(___,mdl_index)` returns a subset of the batch linearization results. `mdl_index` specifies the index of the linearizations of interest, in addition to any of the input arguments in previous syntaxes.

Use this syntax for efficient linearization, when you want to obtain the transfer function for only a subset of the batch linearization results.

`[sys,info] = getIOTransfer(___)` returns additional linearization information.

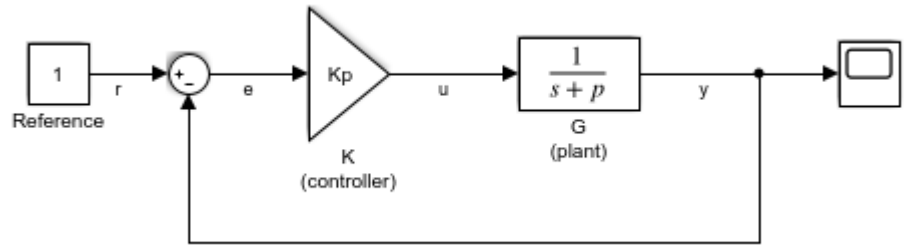
Examples

Obtain Closed-Loop Transfer Function from Reference to Plant Output

Obtain the closed-loop transfer function from the reference signal, `r`, to the plant output, `y`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:
$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the closed-loop transfer function from the reference signal, `r`, to the plant output, `y`, add both points to `sllin`.

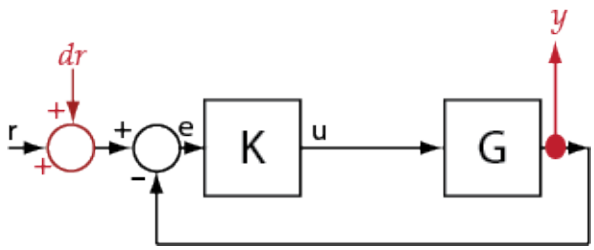
```
addPoint(sllin,{'r','y'});
```

Obtain the closed-loop transfer function from `r` to `y`.

```
sys = getIOTransfer(sllin,'r','y');
tf(sys)
```

```
ans =

  From input "r" to output "y":
    3
  -----
  s + 8

Continuous-time transfer function.
```

The software adds a linearization input at `r`, `dr`, and a linearization output at `y`.



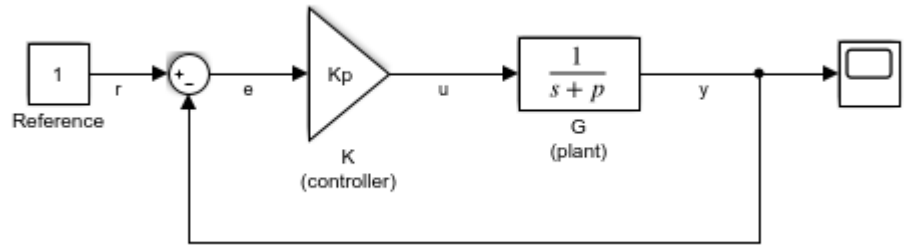
`sys` is the transfer function from `dr` to `y`, which is equal to $(I+GK)^{-1}GK$.

Specify Temporary Loop Opening to Get Plant Model

Obtain the plant model transfer function, `G`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:
$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$.

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the plant model transfer function, use `u` as the input point and `y` as the output point. To eliminate the effects of feedback, you must break the loop. You can break the loop at `u`, `e`, or `y`. For this example, break the loop at `u`. Add these points to `sllin`.

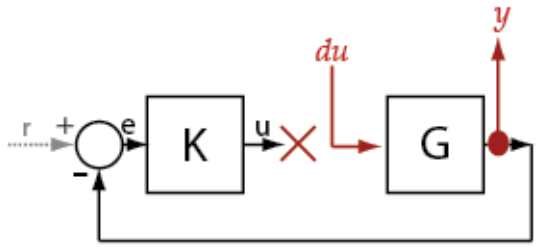
```
addPoint(sllin,{'u','y'});
```

Obtain the plant model transfer function.

```
sys = getIOTransfer(sllin,'u','y','u');
tf(sys)
```

```
ans =

  From input "u" to output "y":
    1
  -----
  s + 5

Continuous-time transfer function.
```

The second input argument specifies `u` as the input, while the fourth input argument specifies `u` as a temporary loop opening.



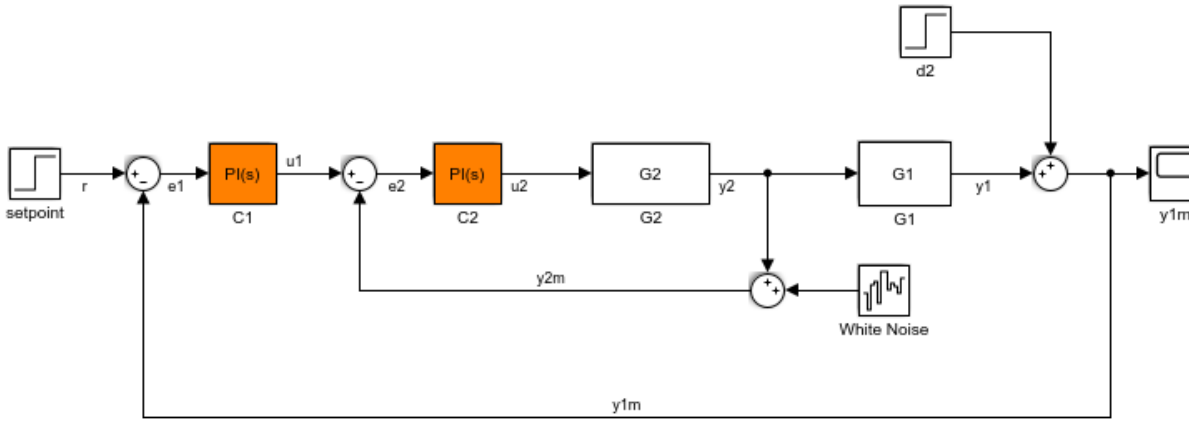
`sys` is the transfer function from `du` to `y`, which is equal to G.

Obtain Open-Loop Response Transfer Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, calculate the open-loop response transfer function for the inner loop, from `e2` to `y2`, for the maximum value of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the open-loop transfer function for the inner loop, use `e2` and `y2` as analysis points. To eliminate the effects of the outer loop, break the loop at `e2`. Add `e2` and `y2` to `sllin` as analysis points.

```
addPoint(sllin,{'e2','y2'})
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

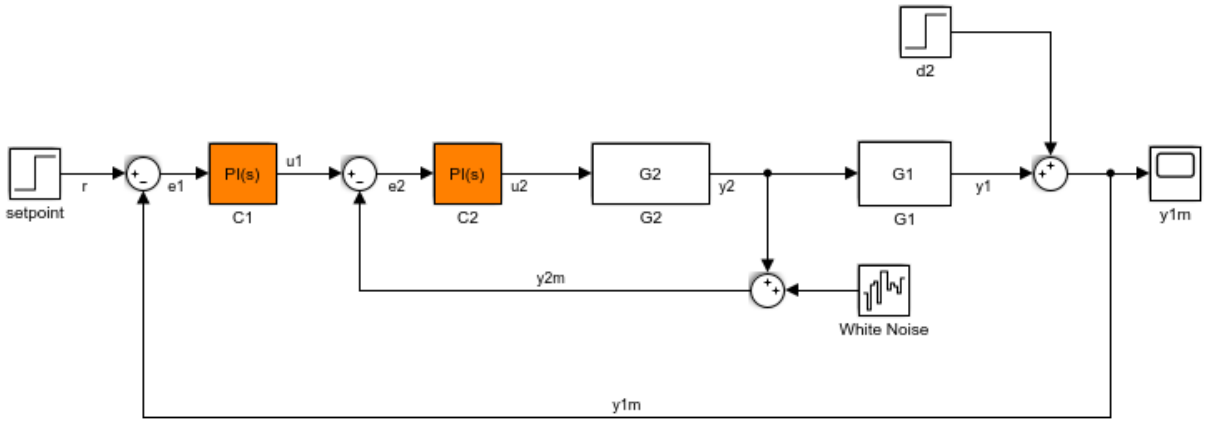
Obtain the open-loop transfer function from `e2` to `y2`.

```
sys = getIOTransfer(sllin,'e2','y2','e2',mdl_index);
```

Obtain Offsets from Input/Output Transfer Function

Open Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add analysis points to calculate the closed-loop transfer function.

```
addPoint(sllin,{'r','y1m'});
```

Calculate the input/output transfer function, and obtain the corresponding linearization offsets.

```
[sys,info] = getIOTransfer(sllin,'r','y1m');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [6x1 double]
           dx: [6x1 double]
```

```
          u: 1
          y: 0
  StateName: {6x1 cell}
  InputName: {'r'}
 OutputName: {'y1m'}
         Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`in` — Input analysis point signal name

character vector | string | cell array of character vectors | string array

Input analysis point on page 13-307 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `in` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `in` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `in` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `in = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `in = {'y1m','y2m'}`.

`out` — Output analysis point signal name

character vector | string | cell array of character vectors | string array

Output analysis point on page 13-307 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `out` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `out` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `out` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `out = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `out = {'y1m','y2m'}`.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is

`'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

For example, `temp_opening = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`ios` — Linearization I/Os

linearization I/O object

Linearization I/Os, created using `linio`, specified as a linearization I/O object.

`ios` must specify signals that are in the list of analysis points for `s`. To view the list of analysis points, type `s`. To use a point that is not in the list of analysis points for `s`, you must first add the point to the list using `addPoint`.

For example:

```
ios(1) = linio('scdcascade/setpoint',1,'input');
ios(2) = linio('scdcascade/Sum',1,'output');
```

Here, `ios(1)` specifies an input, and `ios(2)` specifies an output.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.

- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`sys` — Transfer function for specified I/Os

state-space model

Transfer function for specified I/Os, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `sys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `sys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as a state-space model array of the same size as the parameter grid.

- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as a state-space model array of size `N`-by-`p`.

`info` — Linearization information

structure

Linearization information, returned as a structure with the following field:

`Offsets` — Linearization offsets

`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `sys` is a single state-space model, then `Offsets` is a structure.
- If `sys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `sys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `sys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `sys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `sys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `sys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `sys.InputName`.

Field	Description
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `sys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `sys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see "Approximating Nonlinear Behavior Using an Array of LTI Systems" on page 3-91.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `sys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `sys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `sys`.

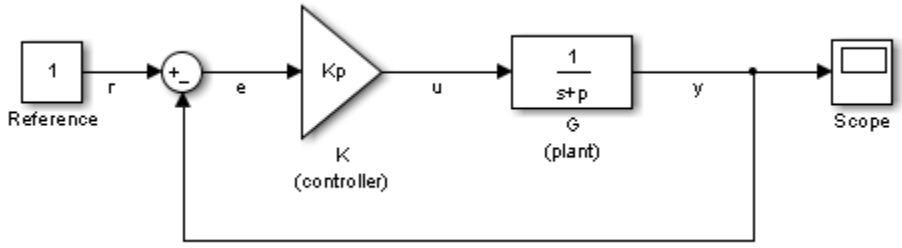
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Definitions

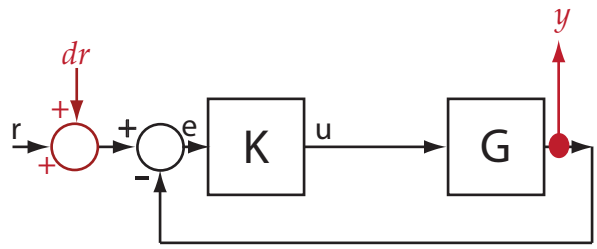
Transfer Functions

A transfer function is an LTI system response at a linearization output point to a linearization input. You perform linear analysis on transfer functions to understand the stability, time-domain characteristics, or frequency-domain characteristics of a system.

You can calculate multiple transfer functions for a given block diagram. Consider the `ex_scd_simple_fdbk` model:



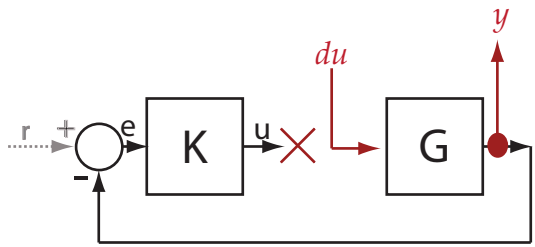
You can calculate the transfer function from the reference input signal to the plant output signal. The reference input (also referred to as setpoint), `r`, originates at the Reference block, and the plant output, `y`, originates at the G block. This transfer function is also called the overall closed-loop transfer function. To calculate this transfer function, the software adds a linearization input at `r`, `dr`, and a linearization output at `y`.



The software calculates the overall closed-loop transfer function as the transfer function from `dr` to `y`, which is equal to $(I+GK)^{-1}GK$.

Observe that the transfer function from `r` to `y` is equal to the transfer function from `dr` to `y`.

You can calculate the plant transfer function from the plant input, `u`, to the plant output, `y`. To isolate the plant dynamics from the effects of the feedback loop, introduce a loop break (or opening) at `y`, `e`, or, as shown, at `u`.



The software breaks the loop and adds a linearization input, `du`, at `u`, and a linearization output at `y`. The plant transfer function is equal to the transfer function from `du` to `y`, which is *G*.

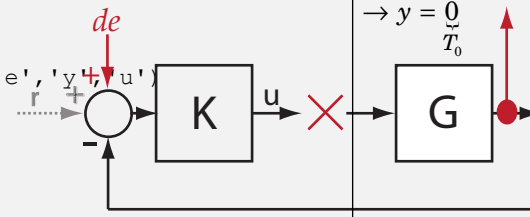
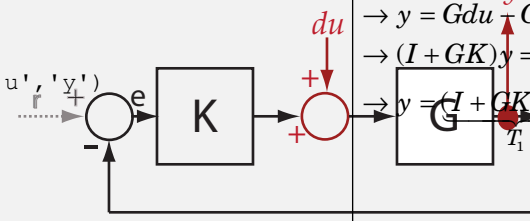
Similarly, to obtain the controller transfer function, calculate the transfer function from the controller input, `e`, to the controller output, `u`. Break the feedback loop at `y`, `e`, or `u`.

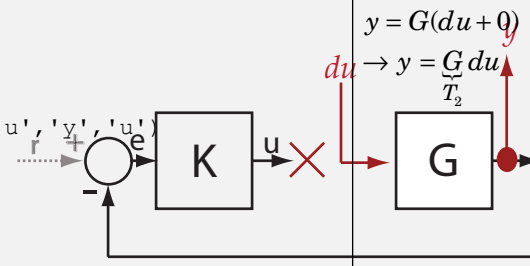
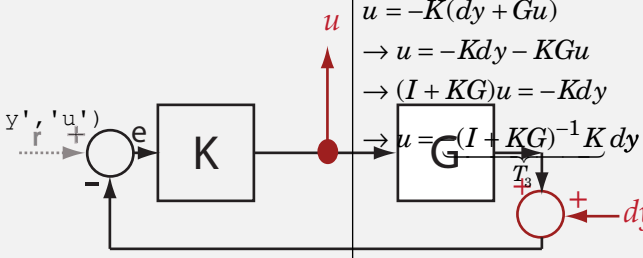
You can use `getIOTransfer` to obtain various open-loop and closed-loop transfer functions. To configure the transfer function, specify analysis points on page 13-307 as inputs, outputs, and openings (temporary or permanent on page 13-308), in any combination. The software treats each combination uniquely. Consider the following code that shows some different ways that you can use the analysis point, `u`, to obtain a transfer function:

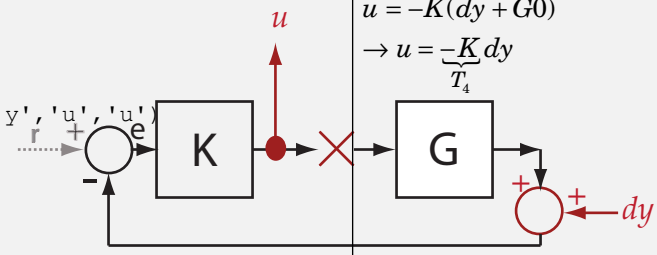
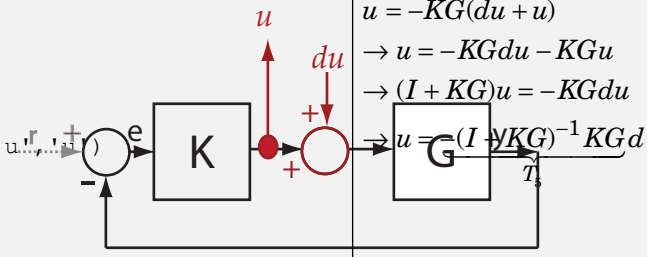
```
sllin = slLinearizer('ex_scd_simple_fdbk')

addPoint(sllin,{'u','e','y'})

T0 = getIOTransfer(sllin,'e','y','u');
T1 = getIOTransfer(sllin,'u','y');
T2 = getIOTransfer(sllin,'u','y','u');
T3 = getIOTransfer(sllin,'y','u');
T4 = getIOTransfer(sllin,'y','u','u');
T5 = getIOTransfer(sllin,'u','u');
T6 = getIOTransfer(sllin,'u','u','u');
```

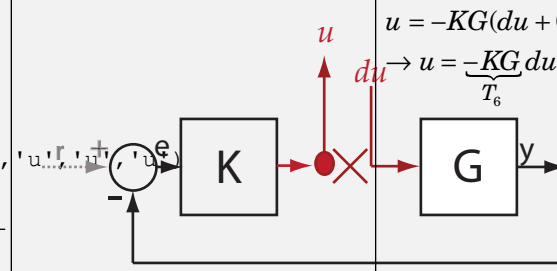
In `T0`, `u` specifies a loop break. In `T1`, `u` specifies only an input, whereas in `T2`, `u` specifies an input and an opening, also referred to as an open-loop input. In `T3`, `u` specifies only an output, whereas in `T4`, `u` specifies an output and an opening, also referred to as an open-loop output. In `T5`, `u` specifies an input and an output, also referred to as a complementary sensitivity point. In `T6`, `u` specifies an input, an output, and an opening,

also referred to as a loop transfer point. The table describes how `getIOTransfer` treats the analysis points, with an emphasis on the different uses of `u`.

u Specifies...	How `getIOTransfer` Treats Analysis Points	Transfer Function
Loop break Example code: `T0 = getIOTransfer(sllin,'e','y','u')`	 The software stops the signal flow at `u`, adds a linearization input, `de`, at `e`, and a linearization output at `y`.	$y = G0$ $\rightarrow y = \frac{0}{T_0}$
Input Example code: `T1 = getIOTransfer(sllin,'u','y')`	 The software adds a linearization input, `du`, at `u`, and a linearization output at `y`.	$y = G(du - Ky)$ $\rightarrow y = Gdu - GKy$ $\rightarrow (I + GK)y = Gdu$ $\rightarrow y = (I + GK)^{-1} G\,du$ $\frac{y}{T_1}$

u Specifies...	How `getIOTransfer` Treats Analysis Points	Transfer Function
Open-loop input Example code: `T2 = getIOTransfer(sllin,'u','y','u')`	 The software breaks the signal flow and adds a linearization input, `du`, at `u`, and a linearization output at `y`.	$y = G(du + 0)$ $\rightarrow y = \underbrace{G}_{T_2} du$
Output Example code: `T3 = getIOTransfer(sllin,'y','u')`	 The software adds a linearization input, `dy`, at `y` and a linearization output at `u`.	$u = -K(dy + Gu)$ $\rightarrow u = -Kdy - KGu$ $\rightarrow (I + KG)u = -Kdy$ $\rightarrow u = \underbrace{-(I + KG)^{-1} K}_{T_3} dy$

u Specifies...	How `getIOTransfer` Treats Analysis Points	Transfer Function
Open-loop output Example code: `T4 = getIOTransfer(sllin,'y','u','u')`	 The software adds a linearization input, `dy`, at `y` and adds a linearization output and breaks the signal flow at `u`.	$u = -K(dy + G0)$ $\rightarrow u = \underbrace{-K}_{T_4} dy$
Complementary sensitivity point Example code: `T5 = getIOTransfer(sllin,'u','u')` **Tip** You also can obtain the complementary sensitivity function using `getCompSensitivity`.	 The software adds a linearization output and a linearization input, `du`, at `u`.	$u = -KG(du + u)$ $\rightarrow u = -KGdu - KGu$ $\rightarrow (I + KG)u = -KGdu$ $\rightarrow u = \underbrace{-(I + KG)^{-1} KG}_{T_5} du$

u Specifies...	How `getIOTransfer` Treats Analysis Points	Transfer Function
Loop transfer function point Example code: `T6 = getIOTransfer(sllin,'u','u',0);` **Tip** You also can obtain the loop transfer function using `getLoopTransfer`.	 The software adds a linearization output, breaks the loop, and adds a linearization input, `du`, at `u`.	$u = -KG(du + 0)$ $\rightarrow u = \underbrace{-KG}_{T_6} du$

The software does not modify the Simulink model when it computes the transfer function.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see "Mark Signals of Interest for Control System Analysis and Design" on page 2-48 and "Mark Signals of Interest for Batch Linearization" on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getLoopTransfer` | `getSensitivity` | `slLinearizer` | `slTuner`

Topics

"How the Software Treats Loop Openings" on page 2-39
"Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32

"Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41
"Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48

Introduced in R2013b

getLoopTransfer

Open-loop transfer function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
sys = getLoopTransfer(s,pt)
sys = getLoopTransfer(s,pt,sign)

sys = getLoopTransfer(s,pt,temp_opening)
sys = getLoopTransfer(s,pt,temp_opening,sign)

sys = getLoopTransfer(___,mdl_index)

[sys,info] = getLoopTransfer(___)
```

Description

`sys = getLoopTransfer(s,pt)` returns the point-to-point open-loop transfer function on page 13-324 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent loop openings on page 13-326 specified for `s` when it calculates `sys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getLoopTransfer` performs multiple linearizations and returns an array of loop transfer functions.

`sys = getLoopTransfer(s,pt,sign)` specifies the feedback sign for computing the open-loop response. By default, `sys` is the positive-feedback open-loop transfer function.

Set `sign` to `-1` to compute the negative-feedback open-loop transfer function for applications that assume the negative-feedback definition of `sys`. Many classical design and analysis techniques, such as the Nyquist or root locus design techniques, use the negative-feedback convention.

The closed-loop sensitivity at `pt` is equal to `feedback(1,sys,sign)`.

`sys = getLoopTransfer(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the loop transfer function of an inner loop, measured at the plant input, with the outer loop open.

`sys = getLoopTransfer(s,pt,temp_opening,sign)` specifies temporary openings and the feedback sign.

`sys = getLoopTransfer(___,mdl_index)` returns a subset of the batch linearization results. `mdl_index` specifies the index of the linearizations of interest, in addition to any of the input arguments in previous syntaxes.

Use this syntax for efficient linearization, when you want to obtain the loop transfer function for only a subset of the batch linearization results.

`[sys,info] = getLoopTransfer(___)` returns additional linearization information.

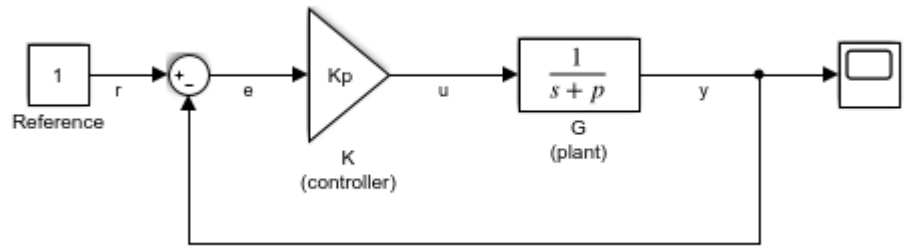
Examples

Obtain Loop Transfer Function at Analysis Point

Obtain the loop transfer function, calculated at `e`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:

$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$.

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the loop transfer function at `e`, add this point to `sllin` as an analysis point.

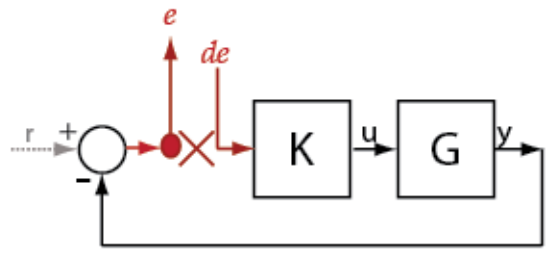
```
addPoint(sllin,'e');
```

Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e');
tf(sys)
```

```
ans =

  From input "e" to output "e":
   -3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



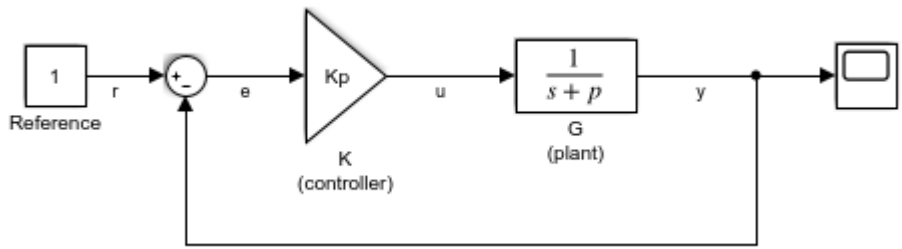
`sys` is the transfer function from `de` to `e`. Because the software assumes positive-feedback, it returns `sys` as $-GK$.

Obtain Negative-Feedback Loop Transfer Function at Analysis Point

Obtain the negative-feedback loop transfer function, calculated at `e`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:
$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$.

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the loop transfer function at `e`, add this point to `sllin` as an analysis point.

```
addPoint(sllin,'e');
```

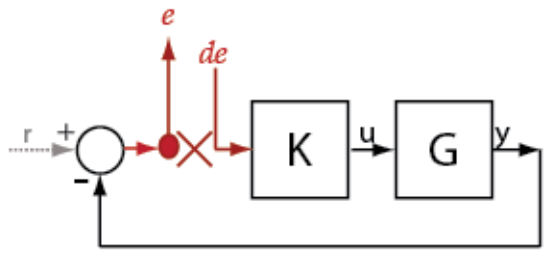
Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e',-1);
tf(sys)
```

```
ans =
```

```
  From input "e" to output "e":
    3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



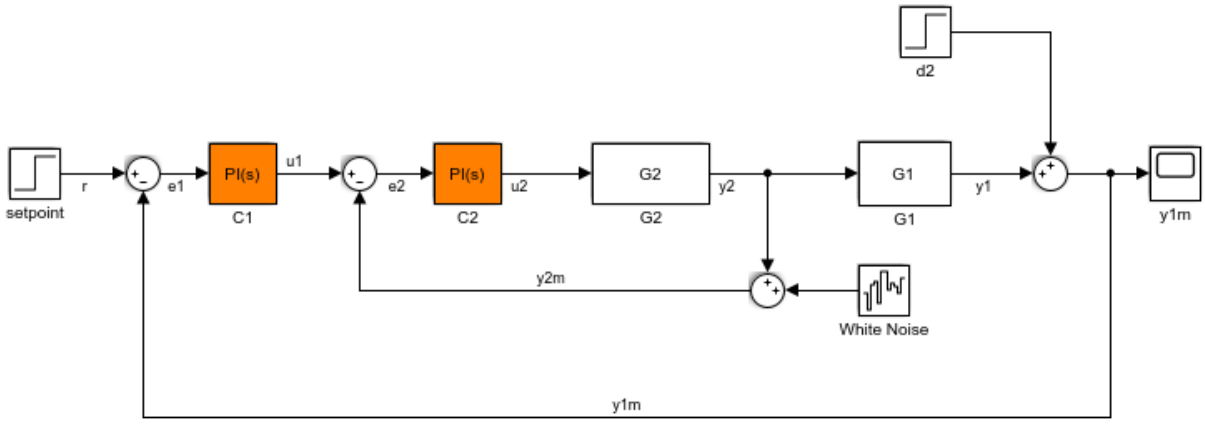
`sys` is the transfer function from `de` to `e`. Because the third input argument indicates negative-feedback, the software returns `sys` as GK.

Specify Temporary Loop Opening for Loop Transfer Function Calculation

Obtain the loop transfer function for the inner loop, calculated at `e2`, for the `scdcascade` model.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the loop transfer function for the inner loop, use the `e2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add these points to `sllin`.

```
addPoint(sllin,{'e2','y1m'});
```

Obtain the inner-loop loop transfer function at `e2`.

```
sys = getLoopTransfer(sllin,'e2','y1m');
```

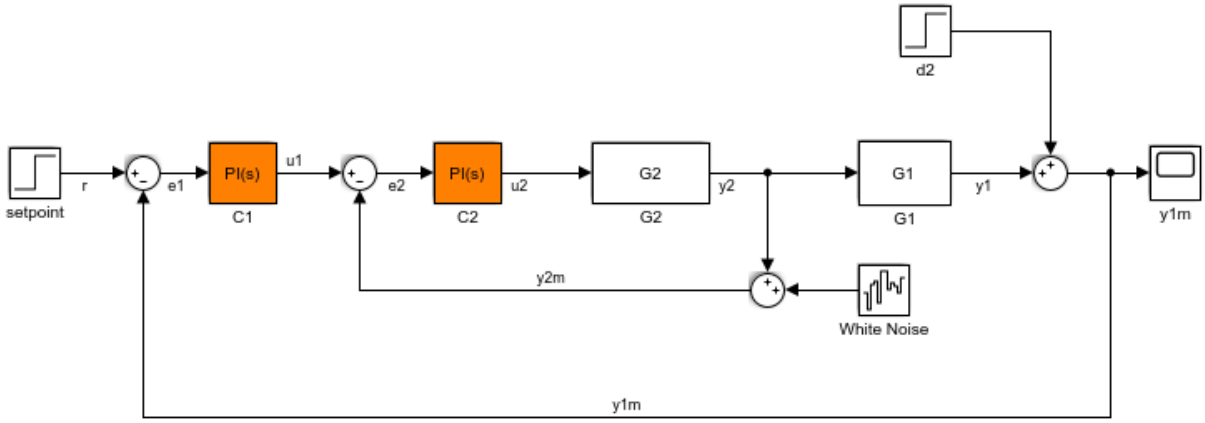
Here, `'y1m'`, the third input argument, specifies a temporary loop opening. The software assumes positive-feedback when it calculates `sys`.

Obtain Loop Transfer Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller, in the 10% range. For this example, calculate the loop transfer function for the inner loop at `e2` for the maximum values of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the loop transfer function for the inner loop, use the `e2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add these points to `sllin`.

```
addPoint(sllin,{'e2','y1m'});
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

Obtain the inner-loop loop transfer function at `e2`, with the outer loop open.

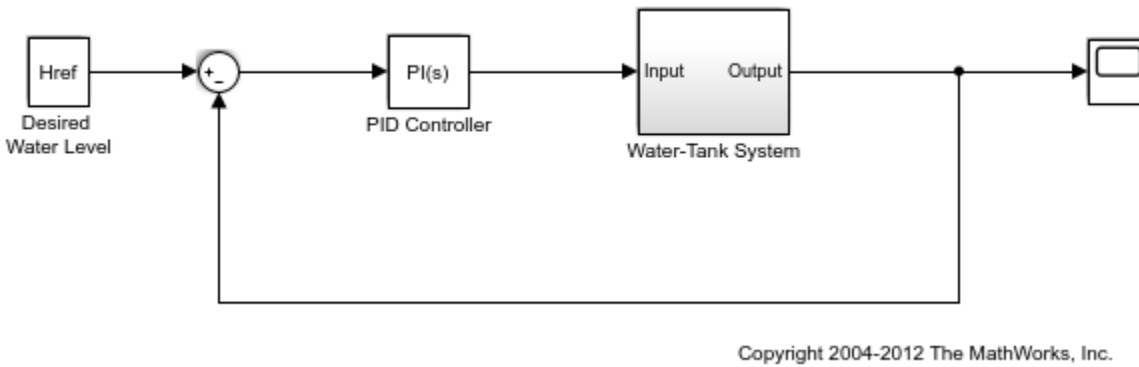
```
sys = getLoopTransfer(sllin,'e2','y1m',-1,mdl_index);
```

The fourth input argument specifies negative-feedback for the loop transfer calculation.

Obtain Offsets from Loop Transfer Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the loop transfer function at the analysis point, and obtain the corresponding linearization offsets.

```
[sys,info] = getLoopTransfer(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [2x1 double]
           dx: [2x1 double]
            u: 1
            y: 1
    StateName: {2x1 cell}
    InputName: {'watertank/Water-Tank System'}
   OutputName: {'watertank/Water-Tank System'}
           Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 13-325 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point

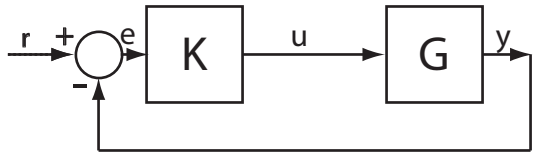
does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

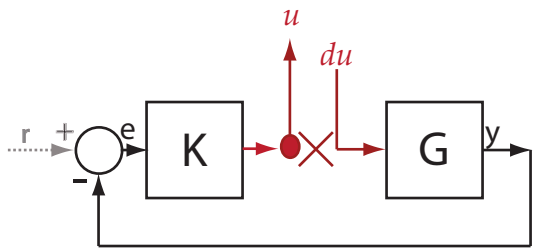
For example, `pt = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `sys`, the software adds a linearization output, followed by a loop break, and then a linearization input at `pt`. Consider the following model:

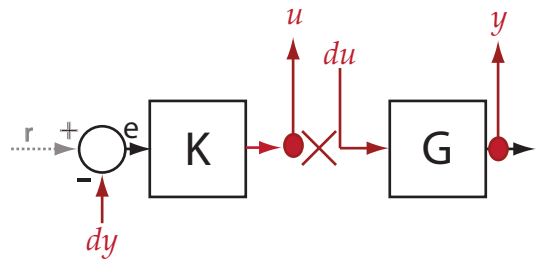


Specify `pt` as `'u'`.



The software computes `sys` as the transfer function from `du` to `u`.

If you specify `pt` as multiple signals, for example `pt = {'u','y'}`, the software adds a linearization output, loop break, and a linearization input at each point.



`du` and `dy` are linearization inputs, and, `u` and `y` are linearization outputs. The software computes `sys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`sign` — Feedback sign

`+1` (default) | `-1`

Feedback sign, specified as one of the following values:

- `+1` (default) — `getLoopTransfer` returns the positive-feedback open-loop transfer function.
- `-1` — `getLoopTransfer` returns the negative-feedback open-loop transfer function. The negative-feedback transfer function is `-1` times the positive-feedback transfer function.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

For example, `temp_opening = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.
- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`sys` — Point-to-point open-loop transfer function

state-space object

Point-to-point open-loop transfer function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `sys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `sys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as a state-space model array of size `N`-by-`p`.

`info` — Linearization information

structure

Linearization information, returned as a structure with the following field:

`Offsets` — Linearization offsets

`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `sys` is a single state-space model, then `Offsets` is a structure.
- If `sys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `sys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `sys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `sys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `sys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `sys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `sys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `sys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `sys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see "Approximating Nonlinear Behavior Using an Array of LTI Systems" on page 3-91.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `sys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.

- If `sys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `sys`.

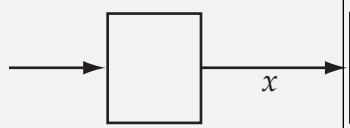
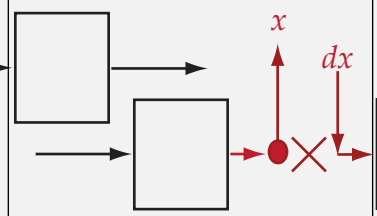
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Definitions

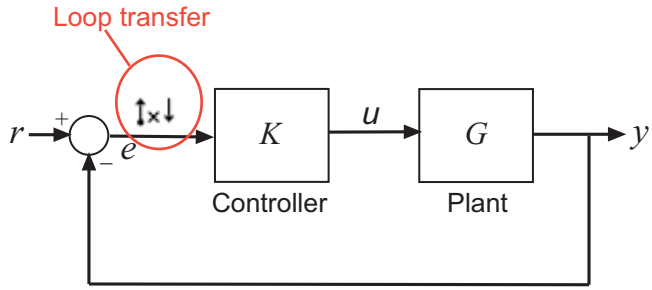
Loop Transfer Function

The loop transfer function at a point is the point-to-point open-loop transfer function from an additive disturbance at a point to a measurement at the same point.

To compute the loop transfer function at an analysis point, `x`, the software adds a linearization output, inserts a loop break, and adds a linearization input, `dx`. The software computes the transfer function from `dx` to `x`, which is equal to the loop transfer function at `x`.

Analysis Point in Simulink Model	How `getLoopTransfer` Interprets Analysis Point	Loop Transfer Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the loop transfer function at `e`:



Here, at `e`, the software adds a linearization output, inserts a loop break, and adds a linearization input, `de`. The loop transfer function at `e`, L, is the transfer function from `de` to `e`. L is calculated as follows:

$$e = \underbrace{-GK}_{L}\, de.$$

To compute $-KG$, use `u` as the analysis point for `getLoopTransfer`.

The software does not modify the Simulink model when it computes the loop transfer function.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see "Mark Signals of Interest for Control System Analysis and Design" on page 2-48 and "Mark Signals of Interest for Batch Linearization" on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getSensitivity` | `slLinearizer` | `slTuner`

Topics

"How the Software Treats Loop Openings" on page 2-39
"Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32
"Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-41
"Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-48

Introduced in R2013b

getOpenings

Get list of openings for `slLinearizer` or `slTuner` interface

Syntax

```
op_names = getOpenings(s)
```

Description

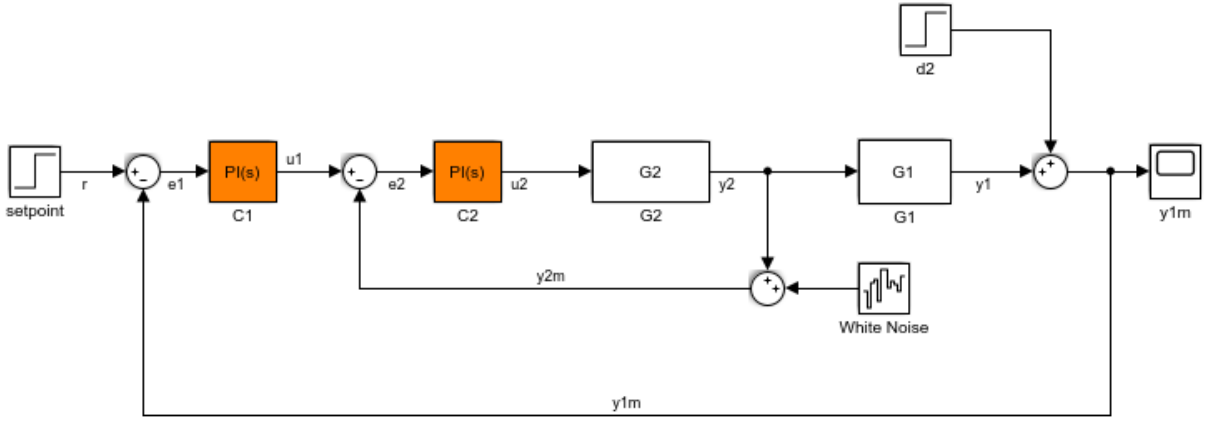
`op_names = getOpenings(s)` returns the names of the permanent openings of `s`, which can be either an `slLinearizer` interface or an `slTuner` interface.

Examples

Obtain Permanent Opening Names of slLinearizer Interface

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface to the model, and add some analysis points to the interface.

```
sllin = slLinearizer(mdl,{'u1','y1'});
```

Suppose you are interested in analyzing only the inner loop. To do so, add `y1m` as a permanent opening of `sllin`.

```
addOpening(sllin,'y1m');
```

In larger models, you may want to open multiple loops to isolate the system of interest.

After performing some additional steps, such as adding more points of interest and extracting transfer functions, suppose you want a list of all the openings of `sllin`.

```
op_names = getOpenings(sllin)
```

```
op_names =

  1x1 cell array
```

```
{'scdcascade/Sum/1[y1m]'}
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Output Arguments

`op_names` — Permanent opening names

cell array of character vectors

Permanent opening names, returned as a cell array of character vectors.

Each entry of `op_names` follows the pattern, `full block path/outport number/[signal name]`.

See Also

`addOpening` | `getIOTransfer` | `removeOpening` | `slLinearizer` | `slTuner`

Introduced in R2014a

getPoints

Get list of analysis points for `slLinearizer` or `slTuner` interface

Syntax

```
pt_names = getPoints(s)
```

Description

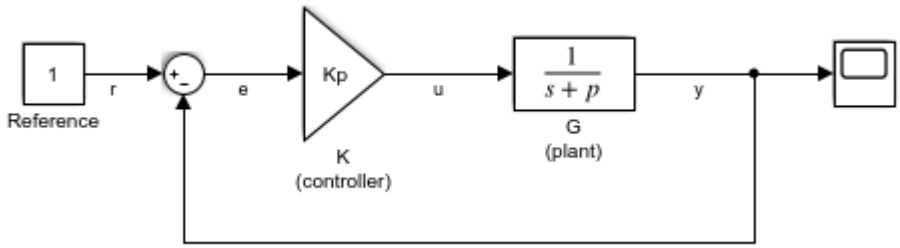
`pt_names = getPoints(s)` returns the names of the analysis points of `s`, which can be either an `slLinearizer` interface or an `slTuner` interface. Use the analysis point names to extract transfer functions using commands such as `getIOTransfer` and to specify tuning goals for an `slTuner` interface.

Examples

Obtain Analysis Point Names of slLinearizer Interface

Open Simulink model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl)
```



Create an `slLinearizer` interface to the model, and add some analysis points to the interface.

```
sllin = slLinearizer(mdl,{'r','e','u','y'});
```

Get the names of all the analysis points associated with `sllin`.

```
pt_names = getPoints(sllin)
```

```
pt_names =

  4x1 cell array

    {'ex_scd_simple_fdbk/Reference/1[r]'     }
    {'ex_scd_simple_fdbk/Sum/1[e]'           }
    {'ex_scd_simple_fdbk/K (controller)/1[u]'}
    {'ex_scd_simple_fdbk/G (plant)/1[y]'     }
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Output Arguments

`pt_names` — Analysis point names

cell array of character vectors

Analysis point names, returned as a cell array of character vectors.

Each entry of `pt_names` follows the pattern, `full block path/outport number/[signal name]`.

See Also

`addPoint` | `getIOTransfer` | `removePoint` | `slLinearizer` | `slTuner`

Topics

"Mark Signals of Interest for Control System Analysis and Design" on page 2-48

Introduced in R2014a

getSensitivity

Sensitivity function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
sys = getSensitivity(s,pt)
sys = getSensitivity(s,pt,temp_opening)
sys = getSensitivity(___,mdl_index)

[sys,info] = getSensitivity(___)
```

Description

`sys = getSensitivity(s,pt)` returns the sensitivity function on page 13-346 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 13-348 specified for `s` when it calculates `sys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getSensitivity` performs multiple linearizations and returns an array of sensitivity functions.

`sys = getSensitivity(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the sensitivity function of an inner loop, with the outer loop open.

`sys = getSensitivity(___,mdl_index)` returns a subset of the batch linearization results. `mdl_index` specifies the index of the linearizations of interest, in addition to any of the input arguments in previous syntaxes.

Use this syntax for efficient linearization, when you want to obtain the sensitivity function for only a subset of the batch linearization results.

`[sys,info] = getSensitivity(___)` returns additional linearization information.

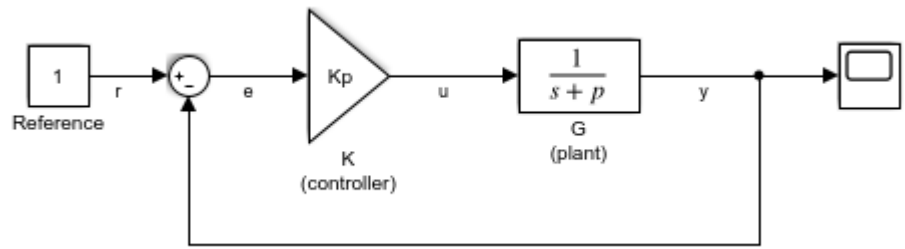
Examples

Sensitivity Function at Analysis Point

For the `ex_scd_simple_fdbk` model, obtain the sensitivity at the plant input, `u`.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



In this model:
$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$.

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the sensitivity at the plant input, `u`, add `u` as an analysis point to `sllin`.

```
addPoint(sllin,'u');
```

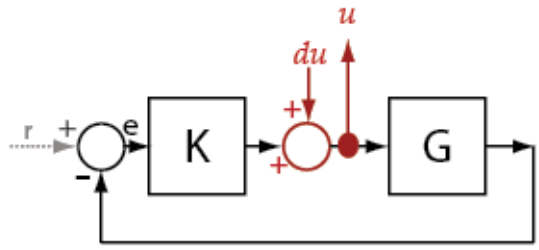
Obtain the sensitivity at the plant input, `u`.

```
sys = getSensitivity(sllin,'u');
tf(sys)
```

```
ans =
```

```
  From input "u" to output "u":
  s + 5
  -----
  s + 8

Continuous-time transfer function.
```

The software uses a linearization input, `du`, and linearization output `u` to compute `sys`.



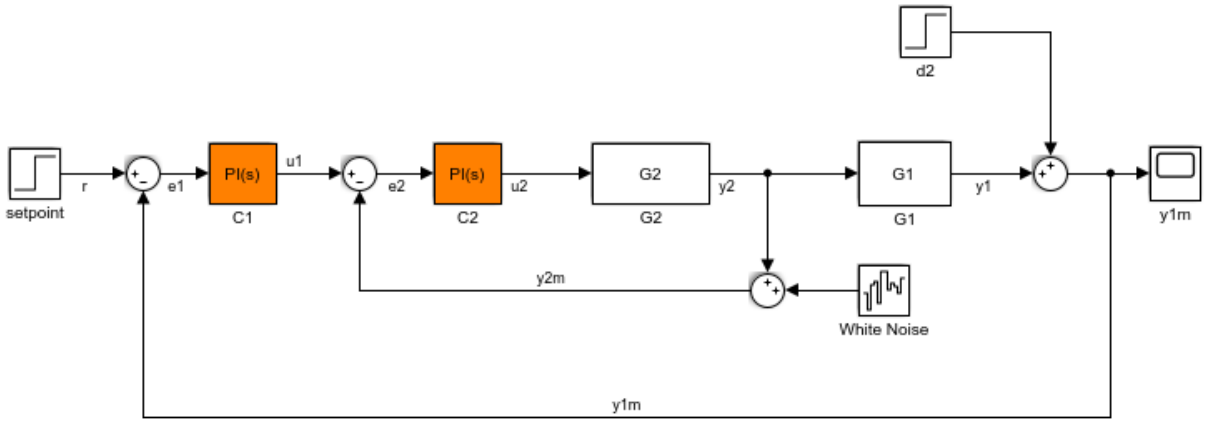
`sys` is the transfer function from `du` to `u`, which is equal to $(I + KG)^{-1}$.

Specify Temporary Loop Opening for Sensitivity Function Calculation

For the `scdcascade` model, obtain the inner-loop sensitivity at the output of `G2`, with the outer loop open.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the sensitivity at the output of `G2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'});
```

Obtain the sensitivity at `y2` with the outer loop open.

```
sys = getSensitivity(sllin,'y2','y1m');
```

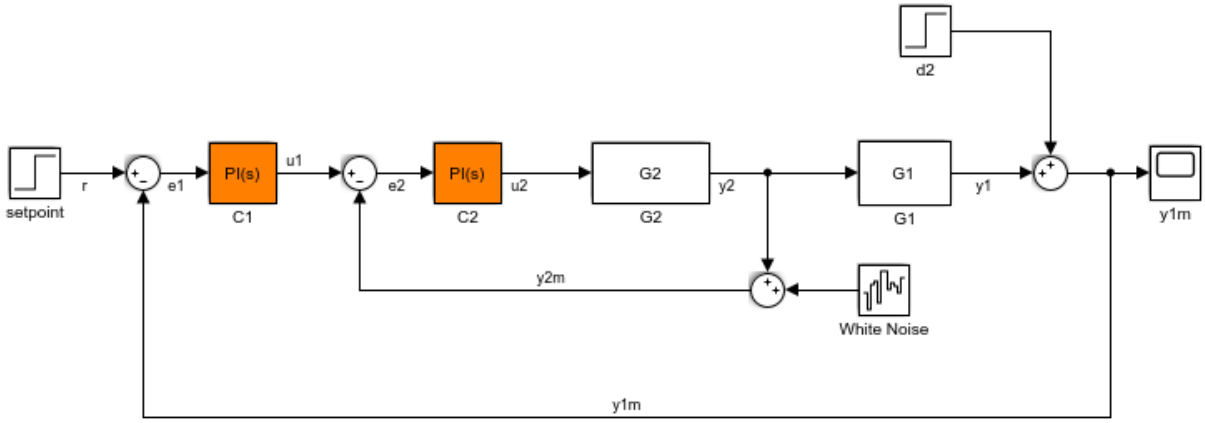
Here, `'y1m'`, the third input argument, specifies a temporary opening of the outer loop.

Obtain Sensitivity Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, obtain the sensitivity at the output of `G2`, with the outer loop open, for the maximum values of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the sensitivity at the output of `G2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin` as analysis points.

```
addPoint(sllin,{'y2','y1m'});
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

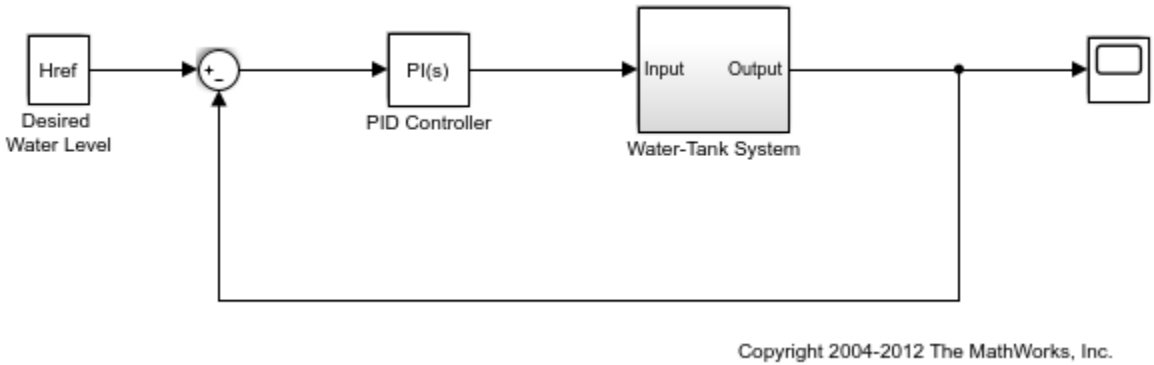
Obtain the sensitivity at the output of `G2` for the specified parameter combination.

```
sys = getSensitivity(sllin,'y2','y1m',mdl_index);
```

Obtain Offsets from Sensitivity Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the sensitivity function at the analysis point, and obtain the corresponding linearization offsets.

```
[sys,info] = getSensitivity(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [2x1 double]
           dx: [2x1 double]
            u: 1
            y: 1
    StateName: {2x1 cell}
    InputName: {'watertank/Water-Tank System'}
   OutputName: {'watertank/Water-Tank System'}
           Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 13-347 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point

does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

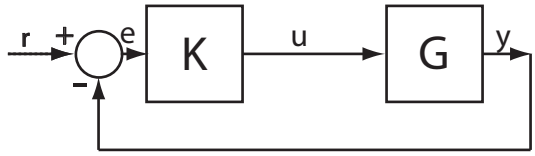
You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

For example, `pt = 'y1m'`.

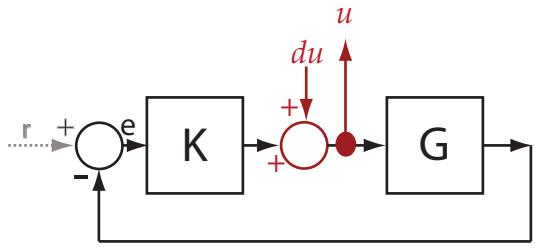
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `sys`, the software adds a linearization input, followed by a linearization output at `pt`.

Consider the following model:

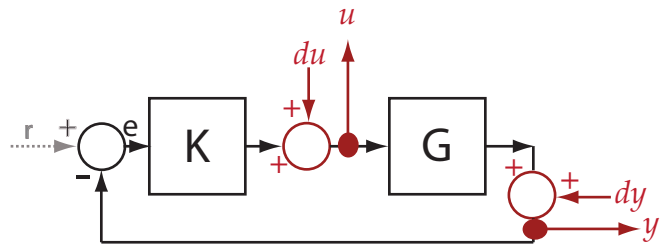


Specify `pt` as `'u'`:



The software computes `sys` as the transfer function from `du` to `u`.

If you specify `pt` as multiple signals, for example `pt = {'u','y'}`, the software adds a linearization input, followed by a linearization output at each point.



`du` and `dy` are linearization inputs, and, `u` and `y` are linearization outputs. The software computes `sys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `temp_opening = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.
- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`sys` — Sensitivity function

state-space model

Sensitivity function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `sys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `sys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as a state-space model array of the same size as the parameter grid.

- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as a state-space model array of size `N`-by-`p`.

`info` — Linearization information

structure

Linearization information, returned as a structure with the following field:

`Offsets` — Linearization offsets

`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `sys` is a single state-space model, then `Offsets` is a structure.
- If `sys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `sys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `sys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `sys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `sys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `sys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `sys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `sys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `sys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see "Approximating Nonlinear Behavior Using an Array of LTI Systems" on page 3-91.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `sys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `sys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `sys`.

`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results

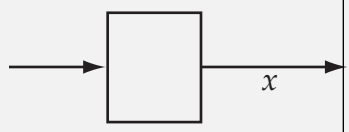
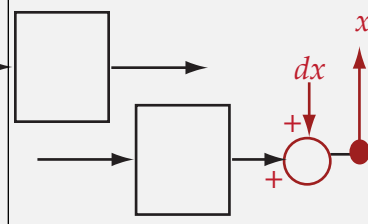
using a `LinearizationAdvisor` object, see "Troubleshoot Linearization Results at Command Line" on page 4-40.

Definitions

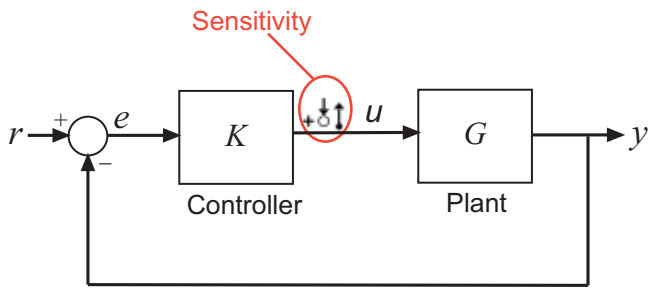
Sensitivity Function

The sensitivity function, also referred to simply as sensitivity, measures how sensitive a signal is to an added disturbance. Sensitivity is a closed-loop measure. Feedback reduces the sensitivity in the frequency band where the open-loop gain is greater than `1`.

To compute the sensitivity at an analysis point, `x`, the software injects a disturbance signal, `dx`, at the point. Then, the software computes the transfer function from `dx` to `x`, which is equal to the sensitivity function at `x`.

Analysis Point in Simulink Model	How `getSensitivity` Interprets Analysis Point	Sensitivity Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the sensitivity function at `u`:



Here, the software injects a disturbance signal (du) at u. The sensitivity at u, S_u, is the transfer function from du to u. The software calculates S_u as follows:

$$u = du - KGu$$
$$\rightarrow (I + KG)u = du$$
$$\rightarrow u = \underbrace{(I + KG)^{-1}}_{S_u} du.$$

Here, I is an identity matrix of the same size as KG.

Similarly, to compute the sensitivity at y, the software injects a disturbance signal (dy) at y. The software computes the sensitivity function as the transfer function from dy to y. This transfer function is equal to $(I+GK)^{-1}$, where I is an identity matrix of the same size as GK.

The software does not modify the Simulink model when it computes the sensitivity transfer function.

Analysis Point

Analysis points, used by the slLinearizer and slTuner interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as getIOTransfer, getLoopTransfer, getSensitivity, and getCompSensitivity. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as systune.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an slLinearizer or slTuner interface, s, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the addPoint command.

To view all the analysis points of s, type s at the command prompt to display the interface contents. For each analysis point of s, the display includes the block name and

port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `slLinearizer` | `slTuner`

Topics

“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-41

“Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-48
“How the Software Treats Loop Openings” on page 2-39

Introduced in R2013b

refresh

Resynchronize `slLinearizer` or `slTuner` interface with current model state

Syntax

```
refresh(s)
```

Description

`refresh(s)` resynchronizes the `slLinearizer` or `slTuner` interface, `s`, with the current state of the model. The interface recompiles the model for the next call to functions that either return transfer functions (such as `getIOTransfer` and `getLoopTransfer`) or functions that tune model parameters (such as `systune` or `looptune`). This model recompilation ensures that the interface uses the current model state when computing linearizations. Block parameterizations and values for tuned blocks are preserved. Use `setBlockParam` to sync blocks with the model.

Use this command after you make changes to the model that impact linearization. Changes that impact linearization include modifying parameter values and reconfiguring blocks and signals.

Examples

Resynchronize slLinearizer Interface with Current Model State

Create an `slLinearizer` interface.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. Then, you linearize the model using the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` commands. The first

time you call one of these commands with `sllin`, the software stores the state of the model in `sllin` and uses it to compute the linearization.

You can change the model after your first call to `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, or `getCompSensitivity` with `sllin`. Some changes impact the linearization, such as changing parameter values. If your change impacts the linearization, call `refresh` to get expected linearization results. For this example, change the proportional gain of the C2 PID controller block.

```
set_param('scdcascade/C2','P','10')
```

Trigger the interface to recompile the model for the next call to `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, or `getCompSensitivity`.

```
refresh(sllin);
```

Resynchronize slTuner Interface with Current Model State

Create an `slTuner` interface.

```
st = slTuner('scdcascade','C2');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. Then, you tune the model block parameters using the `systune` and `looptune` commands. You can also analyze various transfer functions in the model using commands such as `getIOTransfer` and `getLoopTransfer`. The first time you call one of these commands with `st`, the software stores the state of the model in `st` and uses it to compute the linearization.

You can change the model after your first call to one of these commands. Some changes impact the linearization, such as changing parameter values. If your change impacts the linearization, call `refresh` to get expected linearization results. For this example, change the proportional gain of the C1 PID controller block.

```
set_param('scdcascade/C1','P','10')
```

Trigger the interface to recompile the model for the next call to commands such as `getIOTransfer`, `getLoopTransfer`, or `systune`.

```
refresh(st);
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

See Also

`getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `looptune` | `slLinearizer` | `slTuner` | `systune`

Introduced in R2013b

removeAllOpenings

Remove all openings from list of permanent openings in `slLinearizer` or `slTuner` interface

Syntax

```
removeAllOpenings(s)
```

Description

`removeAllOpenings(s)` removes all openings from the list of permanent openings on page 13-354 in the `slLinearizer` or `slTuner` interface, `s`. This function does not modify the Simulink model associated with `s`.

Examples

Remove All Openings from slLinearizer Interface

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add two openings to the interface.

```
addOpening(sllin,{'y2m','y1m'});
```

`'y2m'` and `'y1m'` are the names of two feedback signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of openings for `sllin`.

Remove all the openings from `sllin`.

```
removeAllOpenings(sllin);
```

To verify that all openings have been removed, display the contents of `sllin`, and examine the information about the interface openings.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `removeOpening` | `slLinearizer` | `slTuner`

Introduced in R2013b

removeAllPoints

Remove all points from list of analysis points in `slLinearizer` or `slTuner` interface

Syntax

```
removeAllPoints(s)
```

Description

`removeAllPoints(s)` removes all points from the list of analysis points on page 13-357 for the `slLinearizer` or `slTuner` interface, `s`. This function does not modify the model associated with `s`.

Examples

Remove All Analysis Points

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove all signals from the list of interface analysis points.

```
removeAllPoints(sllin);
```

To verify that all analysis points have been removed, display the contents of `sllin`, and examine the information about the interface analysis points.

```
sllin


slLinearizer linearization interface for "scdcascade":
```

```
No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

See Also

`addPoint` | `removePoint` | `slLinearizer` | `slTuner`

Introduced in R2013b

removeOpening

Remove opening from list of permanent loop openings in `slLinearizer` or `slTuner` interface

Syntax

```
removeOpening(s,op)
```

Description

`removeOpening(s,op)` removes the specified opening, `op`, from the list of permanent openings on page 13-364 for the `slLinearizer` or `slTuner` interface, `s`. You can specify `op` to remove either a single or multiple openings.

`removeOpening` does not modify the model associated with `s`.

Examples

Remove Opening Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Remove the `'y1m'` opening from `sllin`.

```
removeOpening(sllin,'y1m');
```

Remove Multiple Openings Using Signal Names

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Remove the `'y1m'` and `'y2m'` openings from `sllin`.

```
removeOpening(sllin,{'y1m','y2m'});
```

Remove Opening Using Index

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, loop openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Determine the index number of the opening you want to remove. To do this, display the contents of the interface, which includes opening index numbers, in the Command Window.

For this example, remove the `'y1m'` opening from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
3 Permanent openings:
-------------------------
Opening 1:
- Block: scdcascade/Sum3
- Port: 1
- Signal Name: y2m
Opening 2:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m
Opening 3:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1

Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `'y1m'` is the second opening of `sllin` .

Remove the opening from the interface.

```
removeOpening(sllin,2);
```

Remove Multiple Openings Using Index

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

'y2m', 'y1m', and 'u1' are the names of signals in the scdcascade model. The addOpening command adds these signals to the list of permanent openings for sllin.

Determine the index numbers of the openings you want to remove. To do this, display the contents of the interface, which includes opening index numbers, in the Command Window.

For this example, remove the 'y2m' and 'y1m' openings from sllin.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
3 Permanent openings:
--------------------------
Opening 1:
- Block: scdcascade/Sum3
- Port: 1
- Signal Name: y2m
Opening 2:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m
Opening 3:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1

Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that 'y2m' and 'y1m' are the first and second openings of sllin .

Remove the openings from the interface.

```
removeOpening(sllin,[1 2]);
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

op — Opening

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Opening on page 13-364 to remove from the list of permanent openings for `s`, specified as:

- Character vector or string — Opening signal name.

  To determine the signal name associated with an opening, type `s`. The software displays the contents of `s` in the MATLAB command window, including the opening signal names, block names, and port numbers. Suppose an opening does not have a signal name, but only a block name and port number. You can specify `op` as the block name.

  You can specify `op` as a uniquely matching portion of the full signal name or block name. Suppose the full signal name of an opening is `'LoadTorque'`. You can specify `op` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other opening of `s`.

  For example, `op = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple opening names. For example, `op = {'y1m','y2m'}`.
- Positive integer — Opening index.

  To determine the index of an opening, type `s`. The software displays the contents of `s` in the MATLAB command window, including the opening indices. For example, `op = 1`.
- Vector of positive integers — Specifies multiple opening indices. For example, `op = [1 2]`.

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`addOpening` | `removeAllOpenings` | `removePoint` | `slLinearizer` | `slTuner`

Introduced in R2013b

removePoint

Remove point from list of analysis points in `slLinearizer` or `slTuner` interface

Syntax

```
removePoint(s,pt)
```

Description

`removePoint(s,pt)` removes the specified point, `pt`, from the list of analysis points on page 13-369 for the `slLinearizer` or `slTuner` interface, `s`. You can specify `pt` to remove either a single or multiple points.

`removePoint` does not modify the model associated with `s`.

Examples

Remove Analysis Point Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove the `y1m` point from the interface.

```
removePoint(sllin,'y1m');
```

Remove Multiple Analysis Points Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove the `y1m` and `e1` points from the interface.

```
removePoint(sllin,{'y1m','e1'});
```

Remove Analysis Point Using Index

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Determine the index number of the point you want to remove. To do this, display the contents of the interface, which includes analysis point index numbers, in the Command Window.

For this example, remove the `y1m` point from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

3 Analysis points:
--------------------------
Point 1:
- Block: scdcascade/setpoint
- Port: 1
- Signal Name: r
Point 2:
- Block: scdcascade/Sum1
- Port: 1
- Signal Name: e1
Point 3:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
```

```
    BlockSubstitutions : []
    Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `y1m` is the third analysis point of `sllin` .

Remove the point from the interface.

```
removePoint(sllin,3);
```

Remove Multiple Analysis Points Using Index

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Determine the index numbers of the points you want to remove. To do this, display the contents of the interface, which includes analysis point index numbers, in the Command Window.

For this example, remove the `e1` and `y1m` points from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

3 Analysis points:
--------------------------
Point 1:
- Block: scdcascade/setpoint
- Port: 1
- Signal Name: r
Point 2:
- Block: scdcascade/Sum1
- Port: 1
- Signal Name: e1
Point 3:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m

No permanent openings. Use the addOpening command to add new permanent openings.
```

```
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `e1` and `y1m` are the second and third analysis points of `sllin` .

Remove the points from the interface.

```
removePoint(sllin,[2 3]);
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Analysis point on page 13-369 to remove from the list of analysis points for `s`, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose an analysis point does not have a signal name, but only a block name and port number. You can specify `pt` as the block name.

  You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `pt = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.
- Positive integer or — Analysis point index.

  To determine the index of an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis points indices.

  For example, `pt = 1`.
- Vector of positive integers — Specifies multiple analysis point indices. For example, `pt = [1 2]`.

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

See Also

`addPoint` | `removeAllPoints` | `removeOpening` | `slLinearizer` | `slTuner`

Introduced in R2013b

addBlock

Add block to list of tuned blocks for `slTuner` interface

Syntax

```
addBlock(st,blk)
```

Description

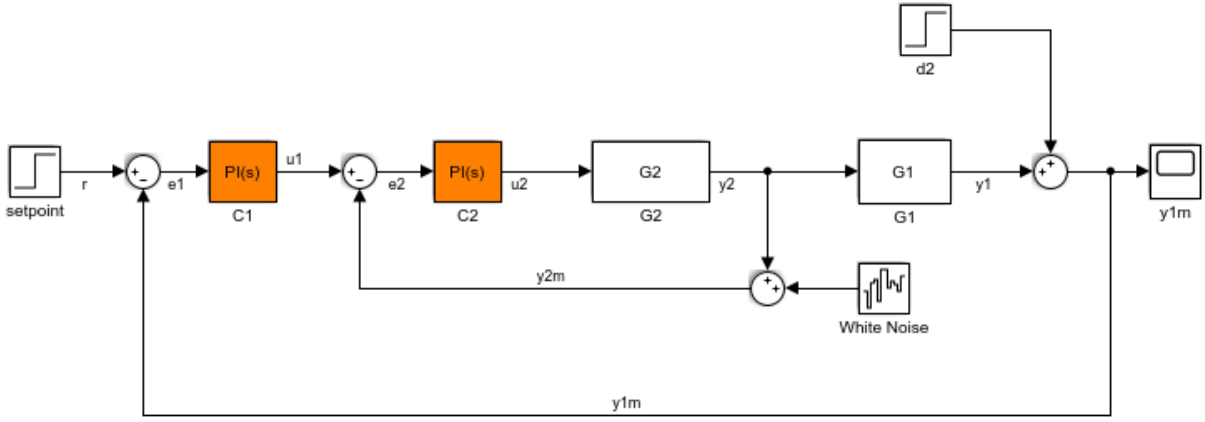
`addBlock(st,blk)` adds the block referenced by `blk` to the list of tuned blocks on page 13-373 of the `slTuner` interface, `st`.

Examples

Add Block to slTuner Interface

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slTuner` interface for the Simulink model, and add a block to the list of tuned blocks of the interface.

```
st = slTuner(mdl,'C1');
addBlock(st,'C2');
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string | cell array of character vectors | string array

Block to add to the list of tuned blocks on page 13-373 for `st`, specified as:

- Character vector or string — Block path. You can specify the full block path or a partial path. The partial path must match the end of the full block path and unambiguously identify the block to add. For example, you can refer to a block by its name, provided the block name appears only once in the Simulink model.

  For example, `blk = 'scdcascade/C1'`.

- Cell array of character vectors or string array — Multiple block paths.

  For example, `blk = {'scdcascade/C1','scdcascade/C2'}`.

Definitions

Tuned Block

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`addOpening` | `addPoint` | `removeBlock` | `slTuner`

Introduced in R2014a

getBlockParam

Get parameterization of tuned block in `slTuner` interface

`getBlockParam` lets you retrieve the current parameterization of a tuned block on page 13-379 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables"Tuned Variables" on page 13-380 for commands such as `systune`.

Syntax

```
blk_param = getBlockParam(st,blk)
[blk_param1,...,blk_paramN] = getBlockParam(st,blk1,...,blkN)

S = getBlockParam(st)
```

Description

`blk_param = getBlockParam(st,blk)` returns the parameterization used to tune the Simulink block, `blk`.

`[blk_param1,...,blk_paramN] = getBlockParam(st,blk1,...,blkN)` returns the parameterizations of one or more specified blocks.

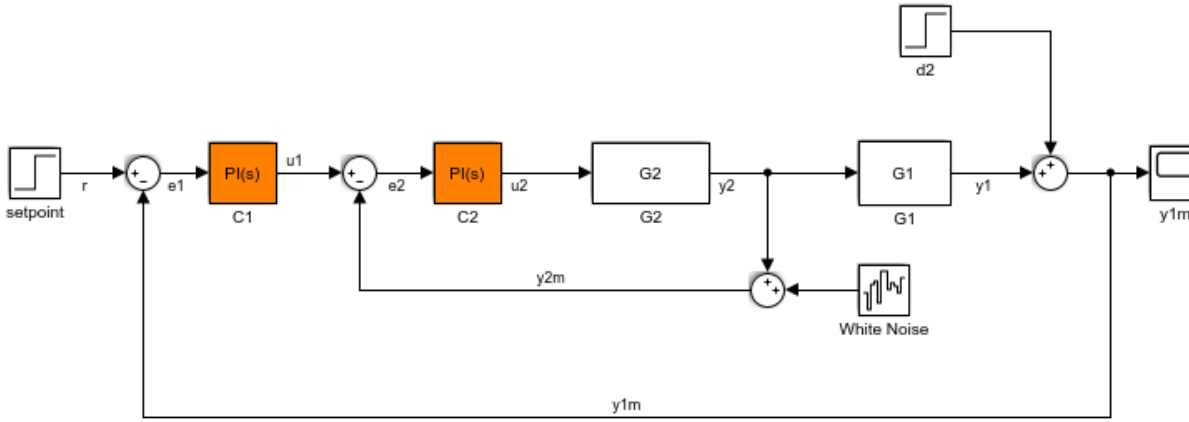
`S = getBlockParam(st)` returns a structure containing the parameterizations of all the tuned blocks of `st`.

Examples

Get Parameterization of Tuned Block

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the block parameterization of one of the tuned blocks.

```
blk_param = getBlockParam(st,'C1')
```

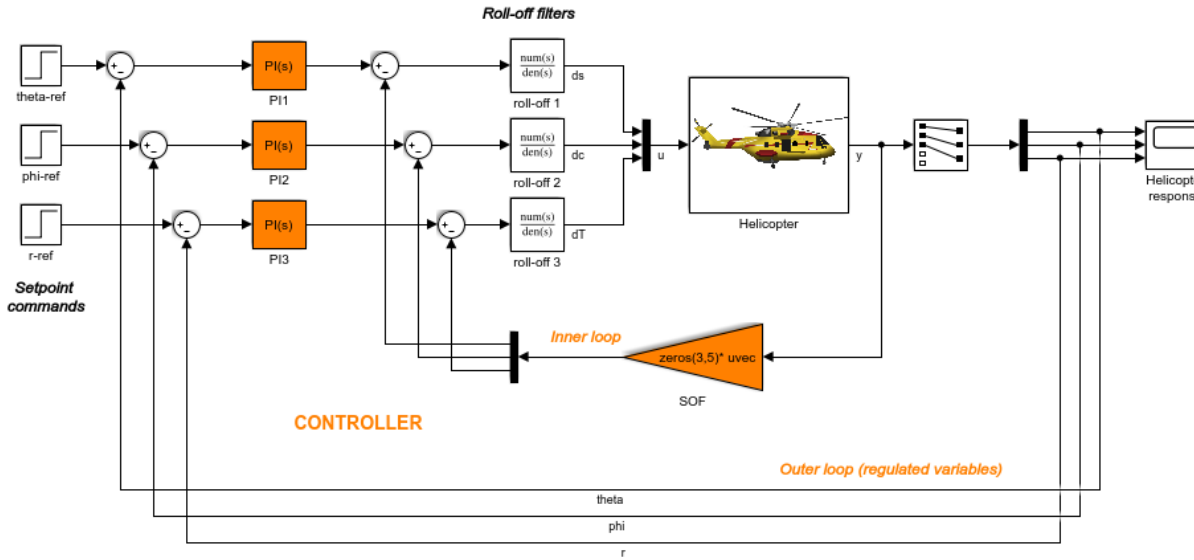
```
blk_param =

  Parametric continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(blk_param)" to see the current value and "get(blk_param)" to see all properti
```

The block `C1` is a PID Controller block. Therefore, its parameterization in `st` is a `tunablePID` Control Design Block.

Get Parameterizations of Multiple Tuned blocks

Create an `slTuner` interface for the `scdhelicopter` model.

```
open_system('scdhelicopter')
st = slTuner('scdhelicopter',{'PI1','PI2','PI3','SOF'});
```



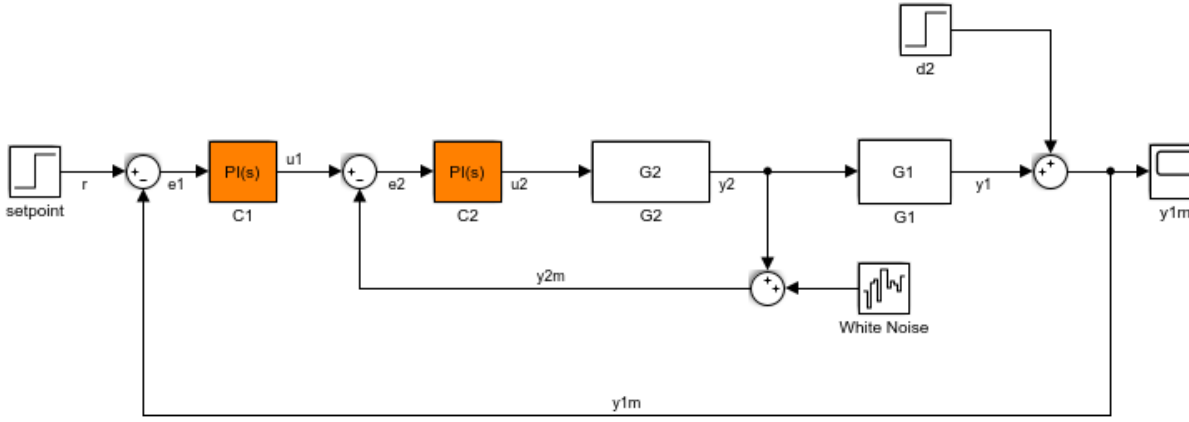
Retrieve the parameterizations for the PI controllers in the model.

```
[parPI1,parPI2,parPI3] = getBlockParam(st,'PI1','PI2','PI3');
```

Get Parameterizations of All Tuned Blocks

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Retrieve the parameterizations for both tuned blocks in `st`.

```
blockParams = getBlockParam(st)
```

```
blockParams =

  struct with fields:

    C1: [1x1 tunablePID]
    C2: [1x1 tunablePID]
```

`blockParams` is a structure with field names corresponding to the names of the tunable blocks in `st`. The field values of `blockParams` are `tunablePID` models, because `C1` and `C2` are both PID Controller blocks.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`blk_param` — Parameterization of tuned block

control design block | generalized model | tunable surface | `[]`

Parameterization of the specified tuned block, returned as one of the following:

- A tunable Control Design Block (Control System Toolbox).
- A tunable `genss` model, tunable `genmat` matrix, or `tunableSurface`, if you specified such a parameterization for `blk` using `setBlockParam`.
- An empty array (`[]`), if `slTuner` cannot parameterize `blk`. You can use `setBlockParam` to specify a parameterization for such blocks.

`S` — Parameterizations of all tuned blocks

structure

Parameterization of all tuned blocks in `st`, returned as a structure. The field names in `S` are the names of the tuned blocks in `st`, and the corresponding field values are block parameterizations as described in `blk_param`.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are

Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or

`setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`genss` | `getBlockValue` | `getTunedValue` | `setBlockParam` | `slTuner` | `tunablePID`

Topics

"How Tuned Simulink Blocks Are Parameterized" (Control System Toolbox)

Introduced in R2011b

getBlockRateConversion

Get rate conversion settings for tuned block in `slTuner` interface

When you use `systune` with Simulink, tuning is performed at the sampling rate specified by the `Ts` property of the `slTuner` interface. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. The rate conversion method associated with each tuned block specifies how this resampling operation should be performed. Use `getBlockRateConversion` to query the block conversion rate and use `setBlockRateConversion` to modify it.

Syntax

```
method = getBlockRateConversion(st,blk)
[method,pwf] = getBlockRateConversion(st,blk)

[IF,DF] = getBlockRateConversion(st,blk)
```

Description

`method = getBlockRateConversion(st,blk)` returns the rate conversion method associated with the tuned block on page 13-385, `blk`.

`[method,pwf] = getBlockRateConversion(st,blk)` also returns the prewarp frequency. When `method` is not `'tustin'`, the prewarp frequency is always 0.

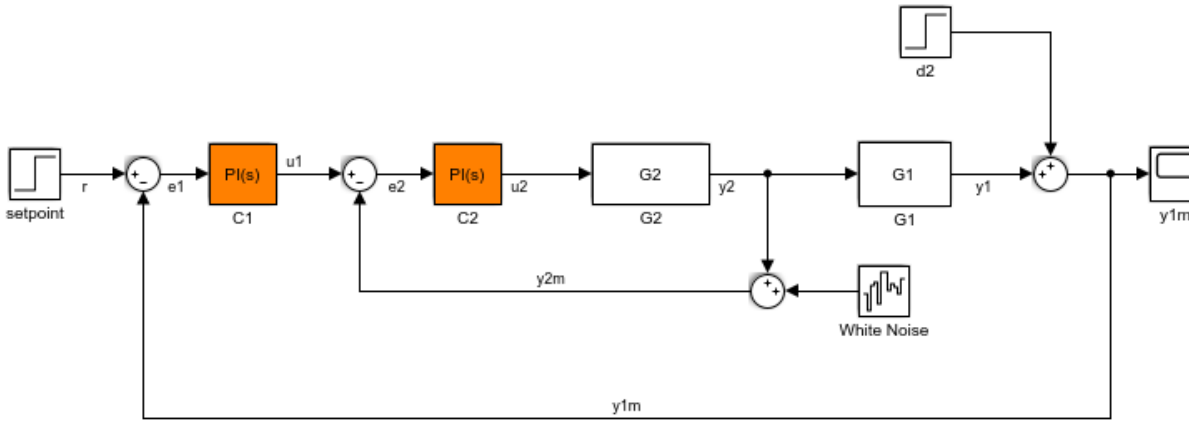
`[IF,DF] = getBlockRateConversion(st,blk)` returns the discretization methods for the integrator and derivative filter terms when `blk` is a PID Controller block.

Examples

Get Rate Conversion Settings of Tuned PID Block

Create an `slTuner` interface for the Simulink model `scdcascade`. Examine the block rate conversion settings of one of the tuned blocks.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```




```
[IF,DF] = getBlockRateConversion(st,'C1')
```

```
IF =

    'Trapezoidal'

DF =

    'Trapezoidal'
```

`C1` is a PID block. Therefore, its rate-conversion settings are expressed in terms of integrator and derivative filter methods. For a continuous-time PID block, the rate-conversion methods are set to `Trapezoidal` by default. To override this setting, use `setBlockRateConversion`.

- “Tuning of a Digital Motion Control System” (Control System Toolbox)

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`method` — Rate conversion method

`'zoh'` | `'foh'` | `'tustin'` | `'matched'`

Rate conversion method associated with `blk`, returned as one of the following:

- `'zoh'` — Zero-order hold on the inputs
- `'foh` — Linear interpolation of inputs
- `'tustin'` — Bilinear (Tustin) approximation
- `'matched'` — Matched pole-zero method (for SISO blocks only)

`pwf` — Prewarp frequency for Tustin method

positive scalar

Prewarp frequency for the Tustin method, returned as a positive scalar.

If the rate conversion method associated with `blk` is zero-order hold or Tustin without prewarp, then `pwf` is 0.

`IF,DF` — Integrator and filter methods

`'ForwardEuler'` | `'BackwardEuler'` | `'Trapezoidal'`

Integrator and filter methods for rate conversion of PID Controller block, each returned as `'ForwardEuler'`, `'BackwardEuler'`, or `'Trapezoidal'`. For continuous-time PID blocks, the default methods are `'Trapezoidal'` for both integrator and derivative filter. For discrete-time PID blocks, `IF` and `DF` are determined by the **Integrator method** and **Filter method** settings in the Simulink block. See the PID Controller and `pid` reference pages for more details about integrator and filter methods.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`setBlockRateConversion` | `slTuner` | `writeBlockValue`

Topics

“Tuning of a Digital Motion Control System” (Control System Toolbox)
“Continuous-Discrete Conversion Methods” (Control System Toolbox)

Introduced in R2014a

getBlockValue

Get current value of tuned block parameterization in `slTuner` interface

`getBlockValue` lets you access the current value of the parameterization of a tuned block on page 13-392 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 13-392 for commands such as `systune`.

Syntax

```
value = getBlockValue(st,blk)
[val1,val2,...] = getBlockValue(st,blk1,blk2,...)

S = getBlockValue(st)
```

Description

`value = getBlockValue(st,blk)` returns the current value of the parameterization of a tunable block, `blk`, in an `slTuner` interface.

`[val1,val2,...] = getBlockValue(st,blk1,blk2,...)` returns the current values of the parameterizations of one or more tuned blocks of `st`.

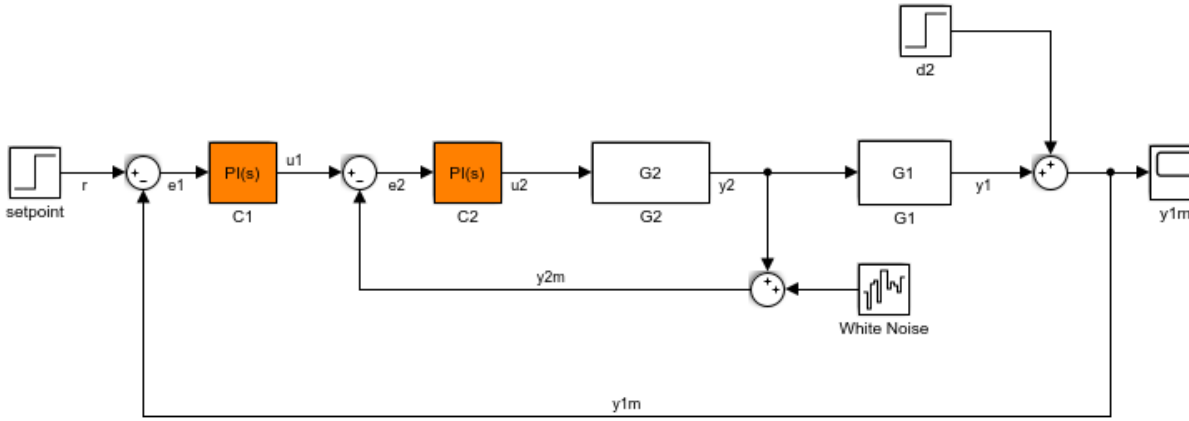
`S = getBlockValue(st)` returns a structure containing the current values of the parameterizations of all tuned blocks of `st`.

Examples

Get Current Value of Tuned Block Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the current parameterization value of one of the tuned blocks.

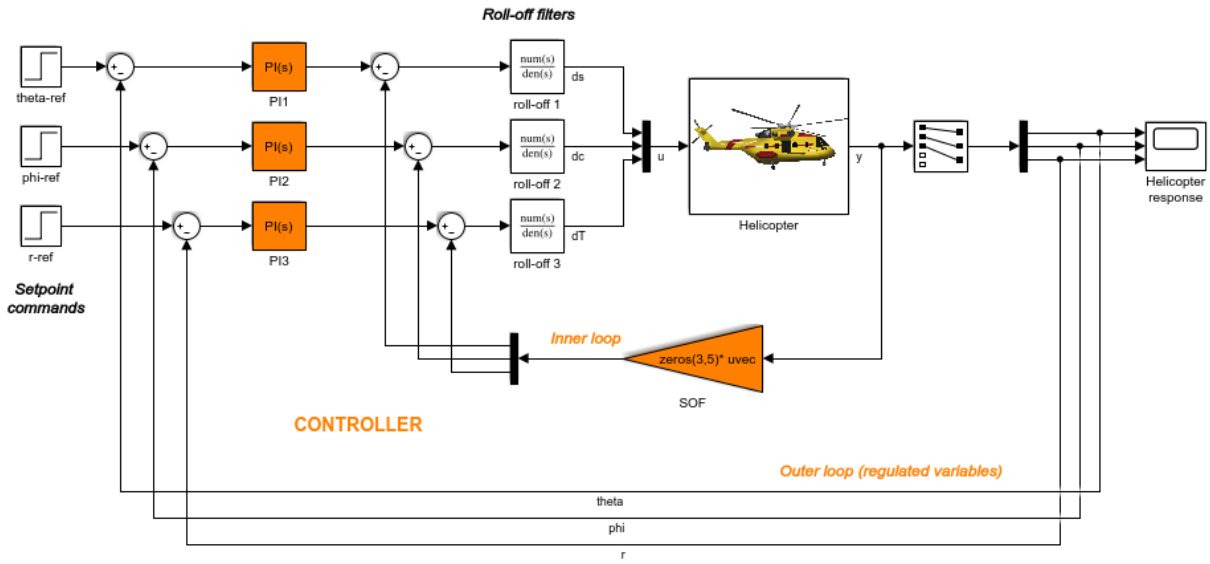
```
val = getBlockValue(st,'C1')
```

```
val =

             1
  Kp + Ki * ---
             s

  with Kp = 0.158, Ki = 0.042

Name: C1
Continuous-time PI controller in parallel form.
```

Get Current Values of Multiple Tuned Block Parameterizations

Create an `slTuner` interface for the `scdhelicopter` model.

```
open_system('scdhelicopter')
st = slTuner('scdhelicopter',{'PI1','PI2','PI3','SOF'});
```



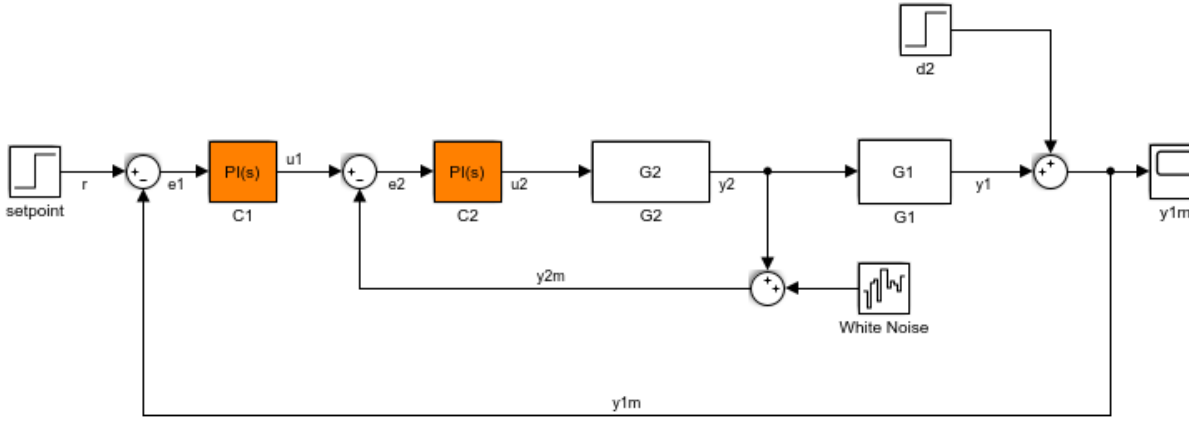
Retrieve the values of parameterizations for the PI controller blocks in the model.

```
[valPI1,valPI2,valPI3] = getBlockParam(st,'PI1','PI2','PI3');
```

Get Current Values of All Tuned Block Parameterizations

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Retrieve the parameterization values for both tuned blocks in `st`.

```
blockValues = getBlockValue(st)
```

```
blockValues =

  struct with fields:

    C1: [1x1 pid]
    C2: [1x1 pid]
```

`blockValues` is a structure with field names corresponding to the names of the tunable blocks in `st`. The field values of `blockValues` are `pid` models, because `C1` and `C2` are both PID Controller blocks.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`value` — Current value of block parameterization

numeric LTI model

Current value of block parameterization, returned as a numeric LTI model (Control System Toolbox), such as `pid`, `ss`, or `tf`.

When the tuning results have not been applied to the Simulink model using `writeBlockValue`, the value returned by `getBlockValue` can differ from the actual Simulink block value.

Note Use `writeBlockValue` to align the block parameterization values with the actual block values in the Simulink model.

`S` — Current values of all block parameterizations

structure

Current values of all block parameterizations in `st`, returned as a structure. The names of the fields in `S` are the names of the tuned blocks in `st`, and the field values are the corresponding numeric LTI models.

You can use this structure to transfer the tuned values from one `slTuner` interface to another `slTuner` interface with the same tuned block parameterizations.

```
S = getBlockValue(st1);
setBlockValue(st2,S);
```

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`getBlockParam` | `getTunedValue` | `setBlockValue` | `slTuner`

Topics

"How Tuned Simulink Blocks Are Parameterized" (Control System Toolbox)

Introduced in R2011b

getTunedValue

Get current value of tuned variable in `slTuner` interface

`getTunedValue` lets you access the current value of a tuned variable on page 13-400 within an `slTuner` interface.

An `slTuner` interface parameterizes each tuned block on page 13-400 as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables for commands such as `systune`.

Syntax

```
value = getTunedValue(st,var)
[value1,value2,...] = getTunedValue(st,var1,var2,...)

S = getTunedValue(st)
```

Description

`value = getTunedValue(st,var)` returns the current value of the tuned variable, `var`, in the `slTuner` interface, `st`.

`[value1,value2,...] = getTunedValue(st,var1,var2,...)` returns the current values of multiple tuned variables.

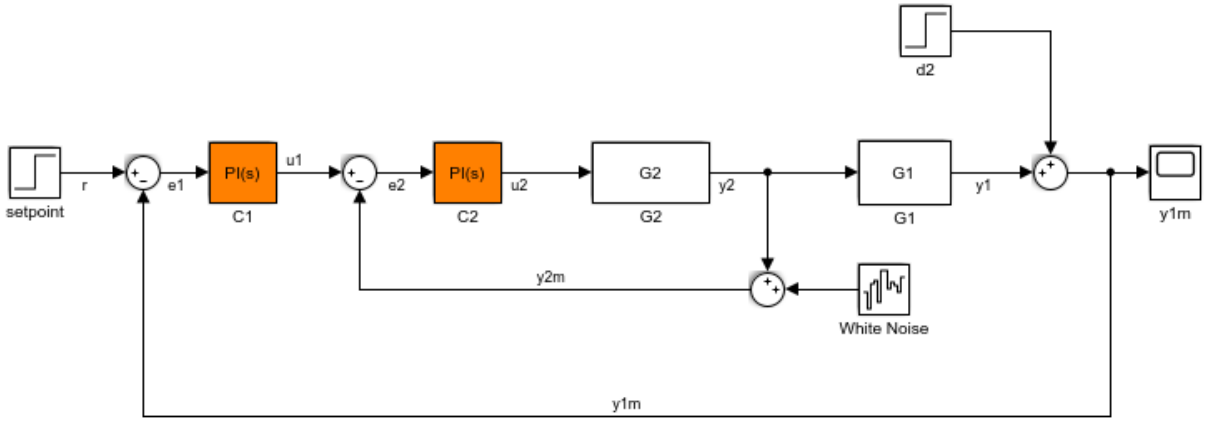
`S = getTunedValue(st)` returns a structure containing the current values of all tuned variables in `st`.

Examples

Query Value of Single Tunable Element within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned value of `Ki`.

```
KiTuned = getTunedValue(st,'Ki')
```

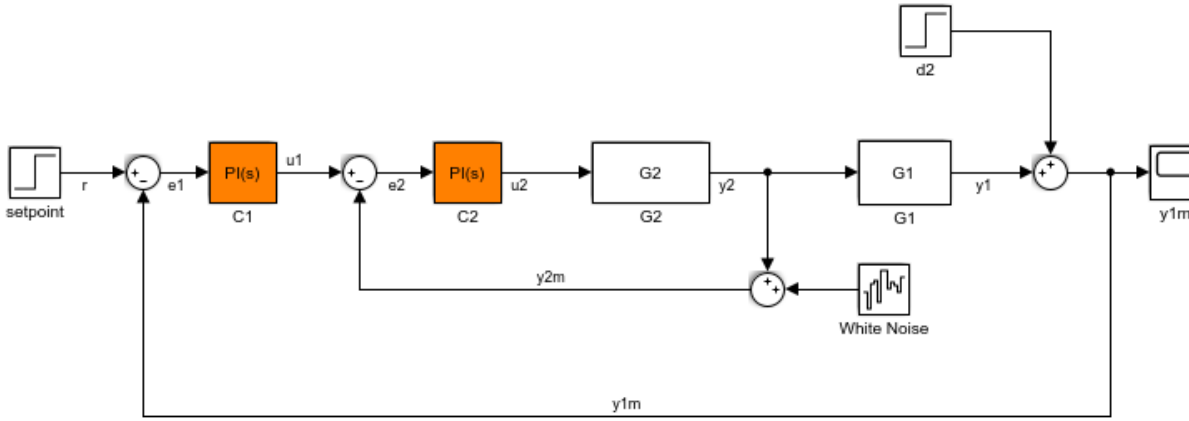
```
KiTuned =

     1
```

To query the value of the tuned block as a whole, `C1`, use `getBlockValue`.

Query Value of Multiple Tunable Elements within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing tunable parameters `Kp` and `Ki`.

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned values of both `Kp` and `Ki`.

```
[KiTuned,KpTuned] = getTunedValue(st,'Ki','Kp')
```

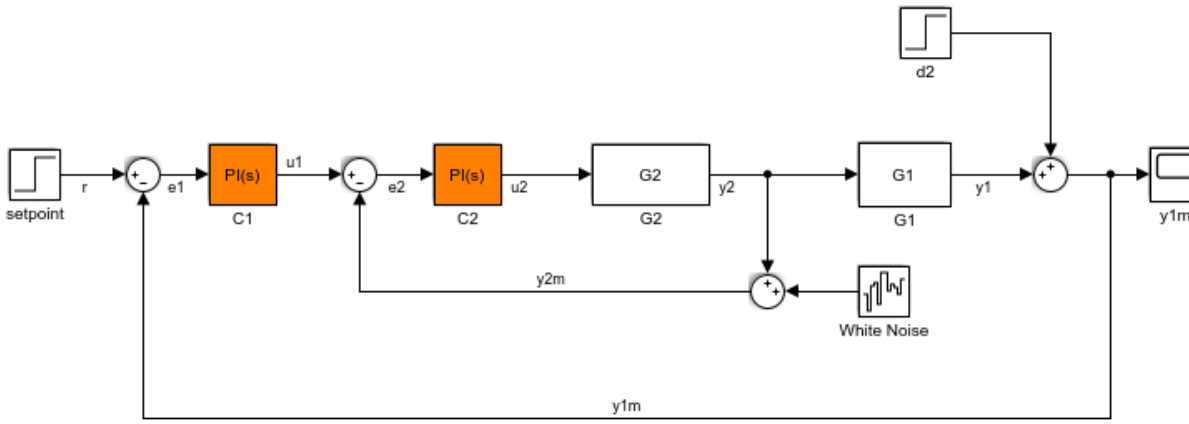
```
KiTuned =

     1
```

```
KpTuned =

     1
```

Query Value of All Tuned Elements in slTuner Interface with Custom Parameterizations

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for tuned block `C1`.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned values of the parameterizations of all the tuned blocks in `st`.

```
S = getTunedValue(st)
```

```
S =
```

```
struct with fields:

  C2: [1x1 pid]
  Ki: 1
  Kp: 1
```

The tuned values are returned in a structure that contains fields for:

- The tuned block, `C2`, which is parameterized as a Control Design Block.
- The tunable elements, `Kp` and `Ki`, within block `C2`, which is parameterized as a custom `genss` model.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`var` — Tuned variable

character vector | string

Tuned variable within `st`, specified as a character vector or string. A tuned variable is any Control Design Block, such `realp`, `tunableSS`, or `tunableGain`, involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. To get a list of all tuned variables within `st`, use `getTunedValue(st)`.

`var` can refer to the following:

- For a block parameterized by a Control Design Block, the name of the block. For example, if the parameterization of the block is

  ```
  C = tunableSS('C')
  ```

  then set `var = 'C'`.

- For a block parameterized by a `genmat`/`genss` model, `M`, the name of any Control Design Block listed in `M.Blocks`. For example, if the parameterization of the block is

  ```
  a = realp('a',1);
  C = tf(a,[1 a]);
  ```

  then set `var = 'a'`.

Output Arguments

`value` — Current value of tuned variable

numeric scalar | numeric array | state-space model

Current value of tuned variable in `st`, returned as a numeric scalar or array or a state-space model. When the tuning results have not been applied to the Simulink model using `writeBlockValue`, the value returned by `getTunedValue` can differ from the Simulink block value.

Note Use `writeBlockValue` to align the block parameterization values with the actual block values in the Simulink model.

`S` — Current values of all tuned variables

structure

Current values of all tuned variables in `st`, returned as a structure. The names of the fields in `S` are the names of the tuned variables in `st`, and the field values are the corresponding numeric scalars or arrays.

You can use this structure to transfer the tuned variable values from one `slTuner` interface to another `slTuner` interface with the same tuned variables, as follows:

```
S = getTunedValue(st1);
setTunedValue(st2,S);
```

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`getBlockParam` | `getBlockValue` | `setTunedValue` | `slTuner` | `tunableSurface`

Topics

"How Tuned Simulink Blocks Are Parameterized" (Control System Toolbox)

Introduced in R2015b

looptune

Tune MIMO feedback loops in Simulink using `slTuner` interface

Syntax

```
[st,gam,info] = looptune(st0,controls,measurements,wc)
[st,gam,info] = looptune(st0,controls,measurements,wc,req1,...,reqN)
[st,gam,info] = looptune(___,opt)
```

Description

`[st,gam,info] = looptune(st0,controls,measurements,wc)` tunes the free parameters on page 13-409 of the control system of the Simulink model associated with the `slTuner` interface, `st0`, to achieve the following goals:

- Bandwidth — Gain crossover for each loop falls in the frequency interval `wc`
- Performance — Integral action at frequencies below `wc`
- Robustness — Adequate stability margins and gain roll-off at frequencies above `wc`

`controls` and `measurements` specify the controller output signals and measurement signals that are subject to the goals, respectively. `st` is the updated `slTuner` interface, `gam` indicates the measure of success in satisfying the goals, and `info` gives details regarding the optimization run.

Tuning is performed at the sample time specified by the `Ts` property of `st0`. For tuning algorithm details, see “Algorithms” on page 13-410.

`[st,gam,info] = looptune(st0,controls,measurements,wc,req1,...,reqN)` tunes the feedback loop to meet additional goals specified in one or more tuning goal objects, `req`. Omit `wc` to drop the default loop shaping goal associated with `wc`. Note that the stability margin goals remain in force.

`[st,gam,info] = looptune(___,opt)` specifies further options, including target gain and phase margins, number of runs, and computation options for the tuning algorithm. Use `looptuneOptions` to create `opt`.

If you specify multiple runs using the `RandomStarts` property of `opt`, `looptune` performs only as many runs required to achieve the target objective value of 1. Note that all tuning goals should be normalized so that a maximum value of 1 means that all design goals are met.

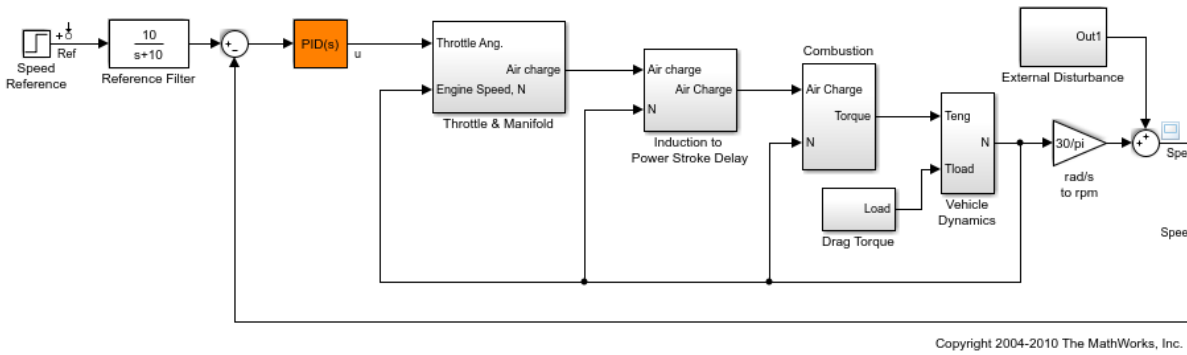
Examples

Tune Controller to Achieve Specified Bandwidth

Tune the PID Controller in the `rct_engine_speed` model to achieve the specified bandwidth.

Open the Simulink model.

```
mdl = 'rct_engine_speed';
open_system(mdl);
```



Create an `slTuner` interface for the model.

```
st0 = slTuner(mdl,'PID Controller');
```

Add the PID Controller output, `u`, as an analysis point to `st0`.

```
addPoint(st0,'u');
```

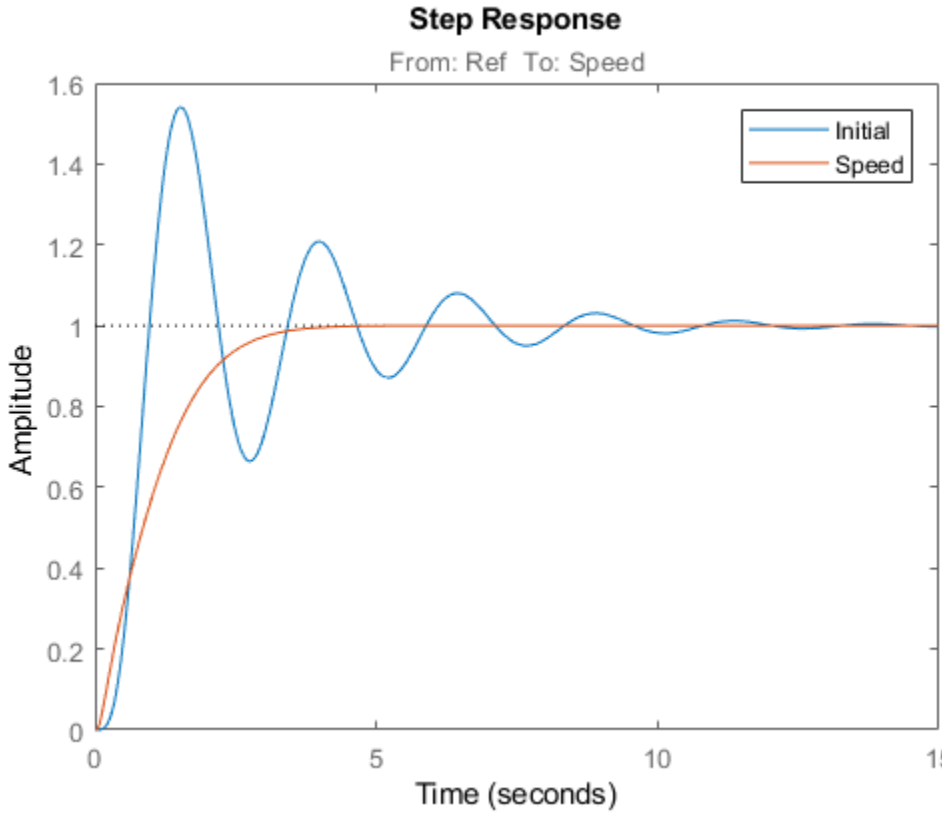
Based on first-order characteristics, the crossover frequency should exceed 1 rad/s for the closed-loop response to settle in less than 5 seconds. So, tune the PID loop using 1 rad/s as the target 0 dB crossover frequency.

```
wc = 1;
st = looptune(st0,'u','Speed',wc);

Final: Peak gain = 0.952, Iterations = 10
Achieved target gain value TargetGain=1.
```

In the call to `looptune`, `'u'` specifies the control signal, and `'Speed'` specifies the measured signal.

Compare the tuned and initial response.

```
stepplot(getIOTransfer(st0,'Ref','Speed'),getIOTransfer(st,'Ref','Speed'));
legend('Initial','Speed');
```



View the tuned block value.

```
showTunable(st)
```

```
Block 1: rct_engine_speed/PID Controller =

                1             s
  Kp + Ki * --- + Kd * --------
                s          Tf*s+1

  with Kp = 0.00141, Ki = 0.00325, Kd = 0.000308, Tf = 0.01

Name: PID_Controller
```

```
Continuous-time PIDF controller in parallel form.
```

To write the tuned values back to the Simulink model, use `writeBlockValue`.

- "Tune Control Systems in Simulink" (Control System Toolbox)
- "Decoupling Controller for a Distillation Column" (Control System Toolbox)
- "Tuning of a Digital Motion Control System" (Control System Toolbox)
- "Tuning of a Two-Loop Autopilot" (Control System Toolbox)

Input Arguments

`st0` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`controls` — Controller output

character vector | cell array of character vectors

Controller output name, specified as one of the following:

- Character vector — Name of an analysis point of `st0`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st0`.

  For example, `'u'`.
- Cell array of character vectors — Multiple analysis point names.

  For example, `{'u','y'}`.

`measurements` — Measurement

character vector | cell array of character vectors

Measurement signal name, specified as one of the following:

- Character vector — Name of an analysis point of `st0`.

You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st0`.

For example, `'u'`.

- Cell array of character vector — Multiple analysis point names.

  For example, `{'u','y'}`.

`wc` — Target crossover region

`[wcmin,wcmax]` | positive scalar

Target crossover region, specified as one of the following:

- `[wcmin,wcmax]` — `looptune` attempts to tune all loops in the control system so that the open-loop gain crosses 0 dB within the target crossover region.
- Positive scalar — Specifies the target crossover region as `[wc/10^0.1,wc*10^0.1]` or `wc` +/- 0.1 decades.

Specify `wc` in the working time units, that is, the time units of the model.

`req1,...,reqN` — Design goals

`TuningGoal` objects

Design goals, specified as one or more `TuningGoal` objects.

For a complete list of the design goals you can specify, see "Tuning Goals" (Control System Toolbox).

`opt` — Tuning algorithm options

options set created using `looptuneOptions`

Tuning algorithm options, specified as an options set created using `looptuneOptions`.

Available options include:

- Number of additional optimizations to run starting from random initial values of the free parameters
- Tolerance for terminating the optimization
- Flag for using parallel processing
- Specification of target gain and phase margin

Output Arguments

`st` — Tuned interface
`slTuner` interface

Tuned interface, returned as an `slTuner` interface.

`gam` — Parameter indicating degree of success at meeting all tuning constraints
scalar

Parameter indicating degree of success at meeting all tuning constraints, returned as a scalar.

A value of `gam <= 1` indicates that all goals are satisfied. A value of `gam >> 1` indicates failure to meet at least one requirement. Use `loopview` to visualize the tuned result and identify the unsatisfied requirement.

For best results, use the `RandomStart` option in `looptuneOptions` to obtain several minimization runs. Setting `RandomStart` to an integer `N > 0` causes `looptune` to run the optimization `N` additional times, beginning from parameter values it chooses randomly. You can examine `gam` for each run to help identify an optimization result that meets your design goals.

`info` — Detailed information about each optimization run
structure

Detailed information about each optimization run, returned as a structure with the following fields:

`Di,Do` — Optimal input and output scalings
state-space models

Optimal input and output scalings, return as state-space models.

The scaled plant is given by `Do\G*Di`.

`Specs` — Design goals used for tuning
vector of `TuningGoal` requirement objects

Design goals used for tuning, returned as a vector of `TuningGoal` requirement objects.

`Runs` — Detailed information about each optimization run
structure

Detailed information about each optimization run, returned as a structure. For details, see “Algorithms” on page 13-410.

The contents of `Runs` are the `info` output of the call to `systune` performed by `looptune`. For information about the fields of `Runs`, see the `info` output argument description on the `systune` reference page.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Algorithms

`looptune` automatically converts target bandwidth, performance goals, and additional design goals into weighting functions that express the goals as an H_∞ optimization problem. `looptune` then uses `systune` to optimize tunable parameters to minimize the H_∞ norm.

For information about the optimization algorithms, see [1].

`looptune` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

See Also

`TuningGoal.Gain` | `TuningGoal.Margins` | `TuningGoal.Tracking` | `addPoint` | `getIOTransfer` | `getLoopTransfer` | `hinfstruct` | `looptune (for genss)` | `looptuneOptions` | `slTuner` | `systune` | `writeBlockValue`

Topics

“Tune Control Systems in Simulink” (Control System Toolbox)
“Decoupling Controller for a Distillation Column” (Control System Toolbox)
“Tuning of a Digital Motion Control System” (Control System Toolbox)
“Tuning of a Two-Loop Autopilot” (Control System Toolbox)
“Structure of Control System for Tuning With looptune” (Control System Toolbox)
“Set Up Your Control System for Tuning with looptune” (Control System Toolbox)

Introduced in R2014a

looptuneSetup

Construct tuning setup for `looptune` to tuning setup for `systune` using `slTuner` interface

Syntax

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)
```

Description

`[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)` converts a tuning setup for `looptune` into an equivalent tuning setup for `systune`. The argument `looptuneInputs` is a sequence of input arguments for `looptune` that specifies the tuning setup. For example,

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,Req1,Req2,loopopt)
```

generates a set of arguments such that `looptune(st0,wc,Req1,Req2,loopopt)` and `systune(st0,SoftReqs,HardReqs,sysopt)` produce the same results.

Use this command to take advantage of additional flexibility that `systune` offers relative to `looptune`. For example, `looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `systune` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `looptune` treats all tuning requirements as soft requirements, optimizing them, but not requiring that any constraint be exactly met. Converting to `systune` allows you to enforce some tuning requirements as hard constraints, while treating others as soft requirements.

You can also use this command to probe into the tuning requirements enforced by `looptune`.

Examples

Convert looptune Problem into systune Problem

Convert a set of `looptune` inputs for tuning a Simulink model into an equivalent set of inputs for `systune`.

Suppose you have created and configured an `slTuner` interface, `st0`, for tuning with `looptune`. Suppose also that you used `looptune` to tune the feedback loop defined in `st0` to within a bandwidth of `wc = [wmin,wmax]`. Convert these variables into a form that allows you to use `systune` for further tuning.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,controls,measurements);
```

The command returns the closed-loop system and tuning requirements for the equivalent `systune` command, `systune(st0,SoftReqs,HardReqs,sysopt)`. The arrays `SoftReqs` and `HardReqs` contain the tuning requirements implicitly imposed by `looptune`. These requirements enforce the target bandwidth and default stability margins of `looptune`.

If you used additional tuning requirements when tuning the system with `looptune`, add them to the input list of `looptuneSetup`. For example, suppose you used a `TuningGoal.Tracking` requirement, `Req1`, and a `TuningGoal.Rejection` requirement, `Req2`. Suppose also that you set algorithm options for `looptune` using `looptuneOptions`. Incorporate these requirements and options into the equivalent `systune` command.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,Req1,Req2,loopopt);
```

The resulting arguments allow you to construct the equivalent tuning problem for `systune`.

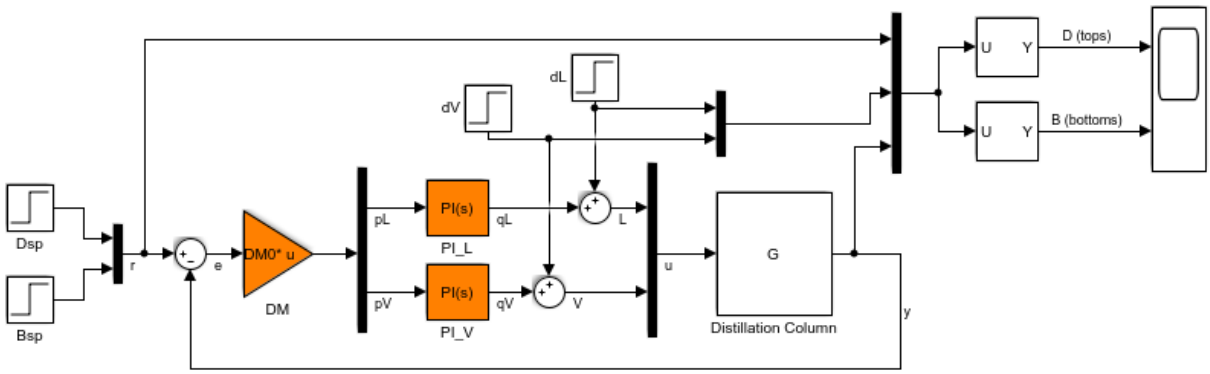
Convert Distillation Column Problem for Tuning With systune

Set up the control system of the Simulink® model `rct_distillation` for tuning with `looptune`. Then, convert the setup to a `systune` problem, and examine the resulting arguments. The results reflect the tuning requirements implicitly enforced when tuning with `looptune`.

Create an `slTuner` interface to the Simulink model, and specify the blocks to be tuned. Configure the interface for tuning with `looptune` by adding analysis points that define

the separation between the plant and the controller. Also add the analysis points needed for imposing tuning requirements.

```
open_system('rct_distillation')

tuned_blocks = {'PI_L','PI_V','DM'};
st0 = slTuner('rct_distillation',tuned_blocks);

addPoint(st0,{'L','V','y','r','dL','dV'});
```



Decoupling controller for a distillation column

This system is now ready for tuning with `looptune`, using tuning goals that you specify. For example, specify a target bandwidth range. Create a tuning requirement that imposes reference tracking in both channels of the system, and a disturbance rejection requirement.

```
wc = [0.1,0.5];
req1 = TuningGoal.Tracking('r','y',15,0.001,1);
max_disturbance_gain = frd([0.05 5 5],[0.001 0.1 10],'TimeUnit','min');
req2 = TuningGoal.Gain({'dL','dV'},'y',max_disturbance_gain);

controls = {'L','V'};
measurement = 'y';

[st,gam,info] = looptune(st0,controls,measurement,wc,req1,req2);

Final: Peak gain = 1.04, Iterations = 79
```

looptune successfully tunes the system to these requirements. However, you might want to switch to systune to take advantage of additional flexibility in configuring your problem. For example, instead of tuning both channels to a loop bandwidth inside wc, you might want to specify different crossover frequencies for each loop. Or, you might want to enforce the tuning requirements, req1 and req2, as hard constraints, and add other requirements as soft requirements.

Convert the looptune input arguments to a set of input arguments for systune.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,controls,measurement,wc,req1,req2);
```

This command returns a set of arguments you can feed to systune for equivalent results to tuning with looptune. In other words, the following command is equivalent to the looptune command.

```
[st,fsoft,ghard,info] = systune(st0,SoftReqs,HardReqs,sysopt);
```

```
Final: Peak gain = 1.04, Iterations = 79
```

Examine the tuning requirements returned by looptuneSetup. When tuning this control system with looptune, all requirements are treated as soft requirements. Therefore, HardReqs is empty. SoftReqs is an array of TuningGoal requirements. These requirements together enforce the bandwidth and margins of the looptune command, plus the additional requirements that you specified.

```
SoftReqs
```

```
SoftReqs =

  5x1 heterogeneous SystemLevel (LoopShape, Tracking, Gain, ...) array with properties:

    Models
    Openings
    Name
```

For example, examine the first entry in SoftReqs.

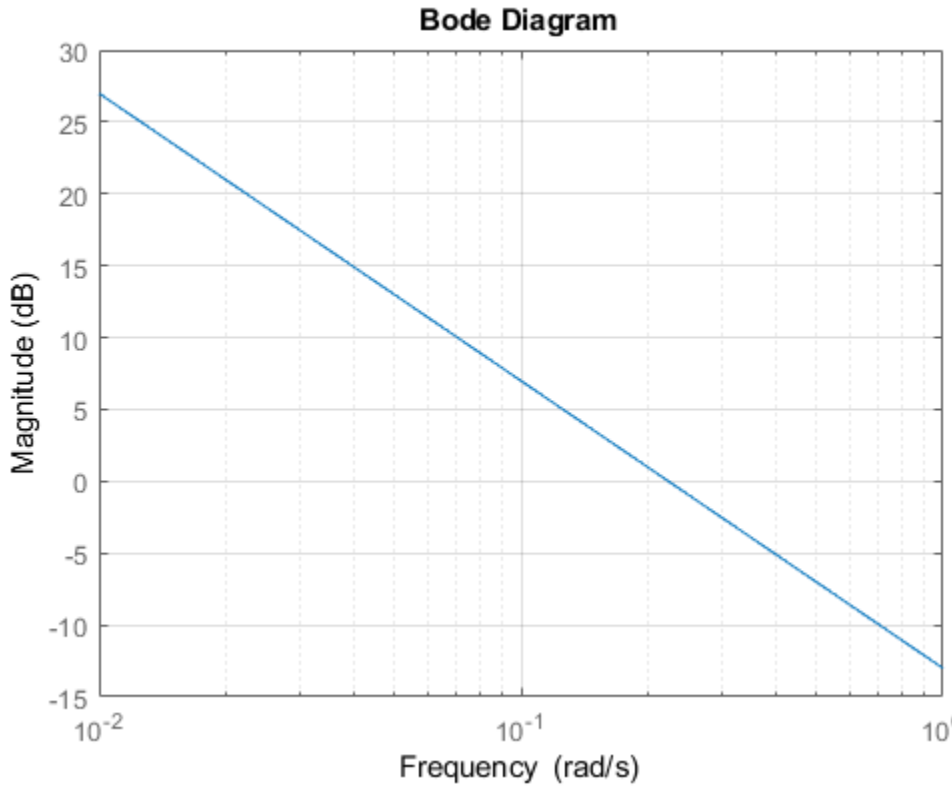
```
SoftReqs(1)
```

```
ans =
```

```
  LoopShape with properties:

       LoopGain: [1x1 zpk]
       CrossTol: 0.3495
          Focus: [0 Inf]
      Stabilize: 1
    LoopScaling: 'on'
       Location: {'y'}
         Models: NaN
       Openings: {0x1 cell}
           Name: 'Open loop GC'
```

`looptuneSetup` expresses the target crossover frequency range `wc` as a `TuningGoal.LoopShape` requirement. This requirement constrains the open-loop gain profile to the loop shape stored in the `LoopGain` property, with a crossover frequency and crossover tolerance (`CrossTol`) determined by `wc`. Examine this loop shape.

```
bodemag(SoftReqs(1).LoopGain,logspace(-2,0)),grid
```



The target crossover is expressed as an integrator gain profile with a crossover between 0.1 and 0.5 rad/s, as specified by `wc`. If you want to specify a different loop shape, you can alter this `TuningGoal.LoopShape` requirement before providing it to `systune`.

`looptune` also tunes to default stability margins that you can change using `looptuneOptions`. For `systune`, stability margins are specified using `TuningGoal.Margins` requirements. Here, `looptuneSetup` has expressed the default stability margins as soft `TuningGoal.Margins` requirements. For example, examine the fourth entry in `SoftReqs`.

```
SoftReqs(4)
```

```
ans =

  Margins with properties:

      GainMargin: 7.6000
     PhaseMargin: 45
    ScalingOrder: 0
           Focus: [0 Inf]
        Location: {2x1 cell}
          Models: NaN
        Openings: {0x1 cell}
            Name: 'Margins at plant inputs'
```

The last entry in `SoftReqs` is a similar `TuningGoal.Margins` requirement constraining the margins at the plant outputs. `looptune` enforces these margins as soft requirements. If you want to convert them to hard constraints, pass them to `systune` in the input vector `HardReqs` instead of the input vector `SoftReqs`.

Input Arguments

`looptuneInputs` — Control system and requirements configured for tuning with `looptune`

valid `looptune` input sequence

Control system and requirements configured for tuning with `looptune`, specified as a valid `looptune` input sequence. For more information about the arguments in a valid `looptune` input sequence, see the `looptune` reference page.

Output Arguments

`st0` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, returned as an `slTuner` interface. `st0` is identical to the `slTuner` interface you use as input to `looptuneSetup`.

`SoftReqs` — Soft tuning requirements

vector of `TuningGoal` requirement objects

Soft tuning requirements for tuning with `systune`, returned as a vector of `TuningGoal` requirement objects.

`looptune` expresses most of its implicit tuning requirements as soft tuning requirements. For example, a specified target loop bandwidth is expressed as a `TuningGoal.LoopShape` requirement with integral gain profile and crossover at the target frequency. Additionally, `looptune` treats all of the explicit requirements you specify (`Req1,...ReqN`) as soft requirements. `SoftReqs` contains all of these tuning requirements.

`HardReqs` — Hard tuning requirements

vector of `TuningGoal` requirement objects

Hard tuning requirements (constraints) for tuning with `systune`, returned as a vector of `TuningGoal` requirement objects.

Because `looptune` treats most tuning requirements as soft requirements, `HardReqs` is usually empty. However, if you change the default `MaxFrequency` option of the `looptuneOptions` set, `loopopt`, then this requirement appears as a hard `TuningGoal.Poles` constraint.

`sysopt` — Algorithm options for `systune` tuning

`systuneOptions` options set

Algorithm options for `systune` tuning, returned as a `systuneOptions` options set.

Some of the options in the `looptuneOptions` set, `loopopt`, are converted into hard or soft requirements that are returned in `HardReqs` and `SoftReqs`. Other options correspond to options in the `systuneOptions` set.

See Also

`looptune` | `looptuneOptions` | `looptuneSetup (for genss)` | `slTuner` | `systune` | `systuneOptions`

Introduced in R2014a

loopview

Graphically analyze results of control system tuning using `slTuner` interface

Syntax

```
loopview(st,controls,measurements)

loopview(st,info)
```

Description

`loopview(st,controls,measurements)` plots characteristics of the control system described by the `slTuner` interface `st`. Use `loopview` to analyze the performance of a tuned control system you obtain using `looptune`.

`loopview` plots:

- The gains of the open-loop frequency response measured at the plant inputs (`controls` analysis points) and at plant outputs (`measurements` analysis points)
- The (largest) gain of the sensitivity and complementary sensitivity functions at the plant inputs or outputs

`loopview(st,info)` uses the `info` structure returned by `looptune` and also plots the target and tuned values of tuning constraints imposed on the system. Use this syntax to assist in troubleshooting when tuning fails to meet all requirements.

Additional plots with this syntax include:

- Normalized multi-loop disk margins (see `loopmargin`) at the plant inputs and outputs
- Target vs. achieved response for any additional tuning goal you used with `looptune`

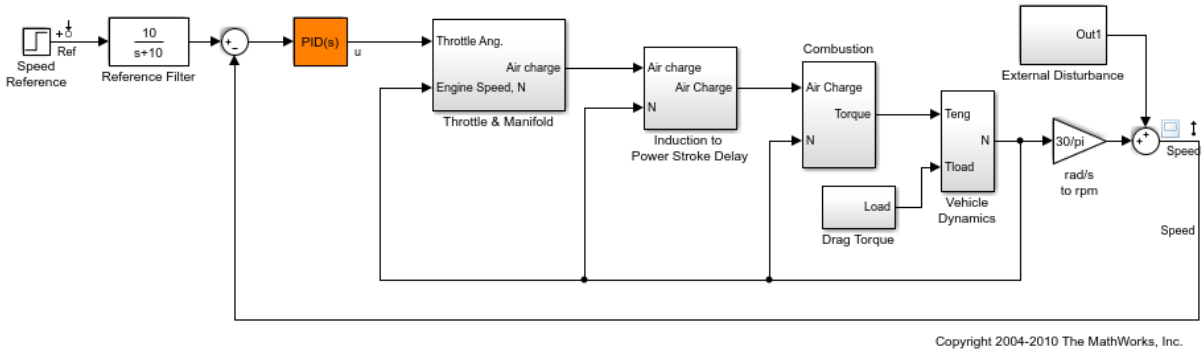
Examples

Graphically Analyze Results of Control System Tuning

Tune the Simulink® model, `rct_engine_speed`, to achieve a specified settling time. Use `loopview` to graphically analyze the tuning results.

Open the model.

```
mdl = 'rct_engine_speed';
open_system(mdl);
```



Create an `slTuner` interface for the model and specify the PID Controller block to be tuned.

```
st0 = slTuner(mdl,'PID Controller');
```

Specify a requirement to achieve a 2 second settling time for the `Speed` signal when tracking the reference signal.

```
req = TuningGoal.Tracking('Ref','Speed',2);
```

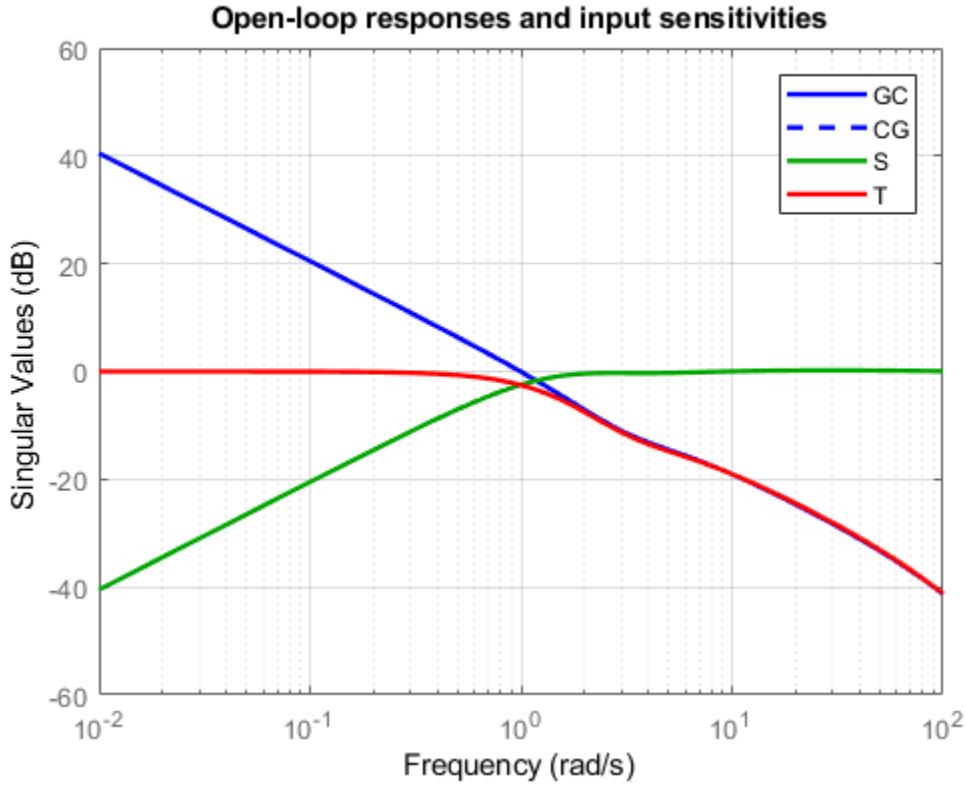
Tune the PID Controller block.

```
addPoint(st0,'u')

control = 'u';
measurement = 'Speed';

wc = 1;

[st1,gam,info] = looptune(st0,control,measurement,wc);
```

```
Final: Peak gain = 0.952, Iterations = 10
Achieved target gain value TargetGain=1.
```

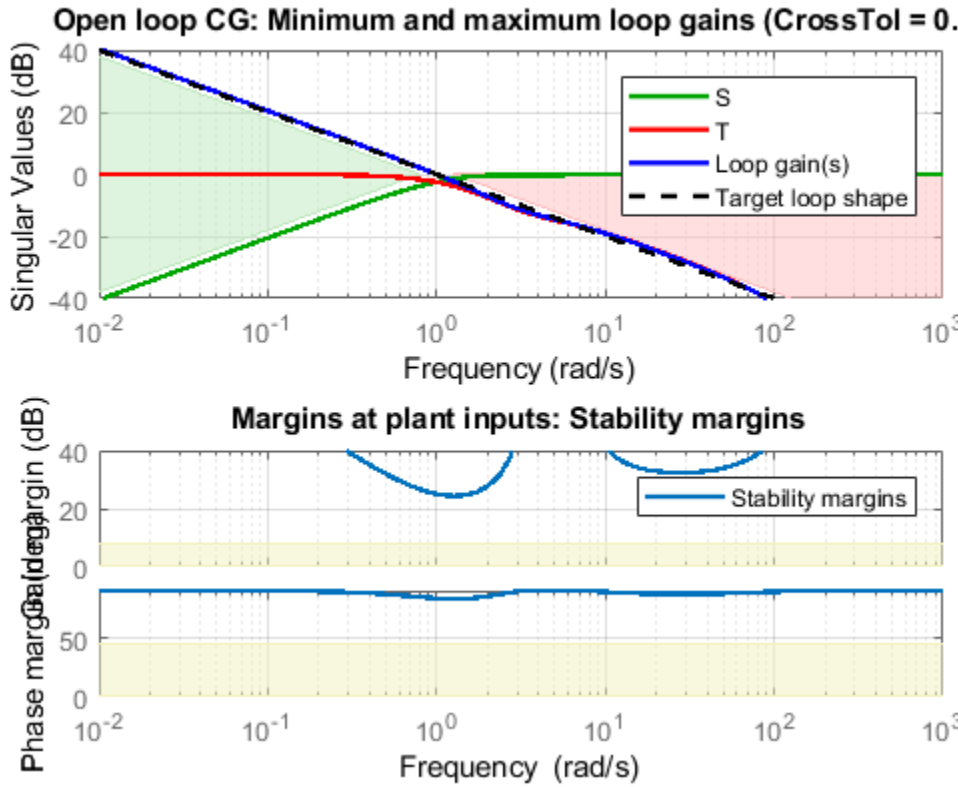
View the response of the model for the tuned block values.

```
loopview(st1,control,measurement);
```



Compare the performance of the tuned block against the tuning goals.

```
figure
loopview(st1,info);
```



- “Decoupling Controller for a Distillation Column” (Control System Toolbox)
- “Tuning of a Two-Loop Autopilot” (Control System Toolbox)
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-48

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`controls` — Controller output
character vector | cell array of character vectors

Controller output name, specified as one of the following:

- Character vector — Name of an analysis point of `st`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st`.

  For example, `'u'`.
- Cell array of character vectors — Multiple analysis point names.

  For example, `{'u','y'}`.

`measurements` — Measurement
character vector | cell array of character vectors

Measurement signal name, specified as one of the following:

- Character vector — Name of an analysis point of `st`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st`.

  For example, `'u'`.
- Cell array of character vector — Multiple analysis point names.

  For example, `{'u','y'}`.

`info` — Detailed information about each optimization run
structure

Detailed information about each optimization run, specified as the structure returned by `looptune`.

Alternative Functionality

For analyzing Control System Toolbox models tuned with `looptune`, use `loopview`.

See Also

`looptune` | `loopview` | `slTuner`

Topics

"Decoupling Controller for a Distillation Column" (Control System Toolbox)
"Tuning of a Two-Loop Autopilot" (Control System Toolbox)
"Mark Signals of Interest for Control System Analysis and Design" on page 2-48

Introduced in R2014a

removeBlock

Remove block from list of tuned blocks in `slTuner` interface

Syntax

```
removeBlock(st,blk)
```

Description

`removeBlock(st,blk)` removes the specified block from the list of tuned blocks on page 13-426 for the `slTuner` interface, `st`. You can specify `blk` to remove either a single or multiple blocks.

`removeBlock` does not modify the Simulink model associated with `st`.

Examples

Remove Block From List of Tuned Blocks of slTuner Interface

Create an `slTuner` interface for the `scdcascade` model. Add `C1` and `C2` as tuned blocks to the interface.

```
st = slTuner('scdcascade',{'C1','C2'});
```

Remove `C1` from the list of tuned blocks of `st`.

```
removeBlock(st,'C1');
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Block to remove from the list of tuned blocks on page 13-426 for `st`, specified as one of the following:

- Character vector or string — Full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`. For example, `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array — Specifies multiple blocks. For example, `blk = {'C1','C2'}`.
- Positive integer — Block index. For example, `blk = 1`.
- Vector of positive integers — Specifies multiple block indices. For example, `blk = [1 2]`.

To determine the name or index associated with a tuned block, type `st`. The software displays the contents of `st` in the MATLAB command window, including the tuned block names.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`addBlock` | `addOpening` | `addPoint` | `slTuner`

Introduced in R2014a

setBlockParam

Set parameterization of tuned block in `slTuner` interface

`setBlockParam` lets you override the default parameterization for a tuned block on page 13-433 in an `slTuner` interface. You can also specify the parameterization for non-atomic components such as Subsystem or S-Function blocks.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 13-433 for commands such as `systune`.

Syntax

```
setBlockParam(st,blk,tunable_mdl)
setBlockParam(st,blk1,tunable_mdl1,blk2,tunable_mdl2,...,blkN,
tunable_mdlN)

setBlockParam(st,blk)
setBlockParam(st)
```

Description

`setBlockParam(st,blk,tunable_mdl)` assigns a tunable model as the parameterization of the specified block of an `slTuner` interface.

`setBlockParam(st,blk1,tunable_mdl1,blk2,tunable_mdl2,...,blkN, tunable_mdlN)` assigns parameterizations to multiple blocks at once.

`setBlockParam(st,blk)` reverts to the default parameterization for the block referenced by `blk` and initializes the block with the current block value in Simulink.

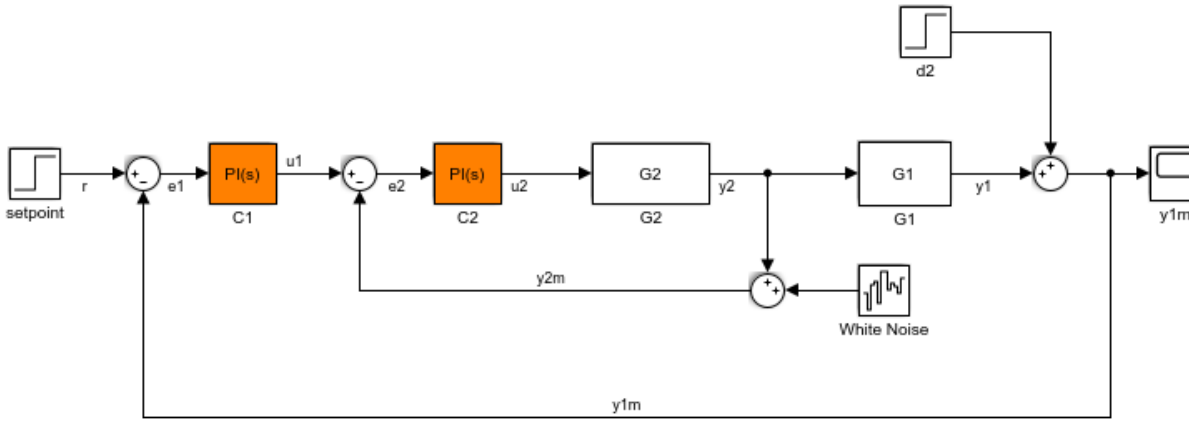
`setBlockParam(st)` reverts all the tuned blocks of `st` to their default parameterizations.

Examples

Set Parameterization of Tuned Block

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Both `C1` and `C2` are PI controllers. Examine the default parameterization of `C1`.

```
getBlockParam(st,'C1')
```

```
ans =

  Parametric continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

The default parameterization is a tunable PI controller (`tunablePID`).

Reparameterize `C1` as a proportional controller. Initialize the proportional gain to 4.2, and assign the parameterization to the block.

```
G = tunableGain('C1',4.2);
setBlockParam(st,'C1',G);
```

Tuning commands, such as `systune`, now use this proportional controller parameterization of the `C1` block of `st`. The custom parameterization is compatible with the default parameterization of the Simulink® block. Therefore, you can use `writeBlockValue` to write the tuned values back to the block.

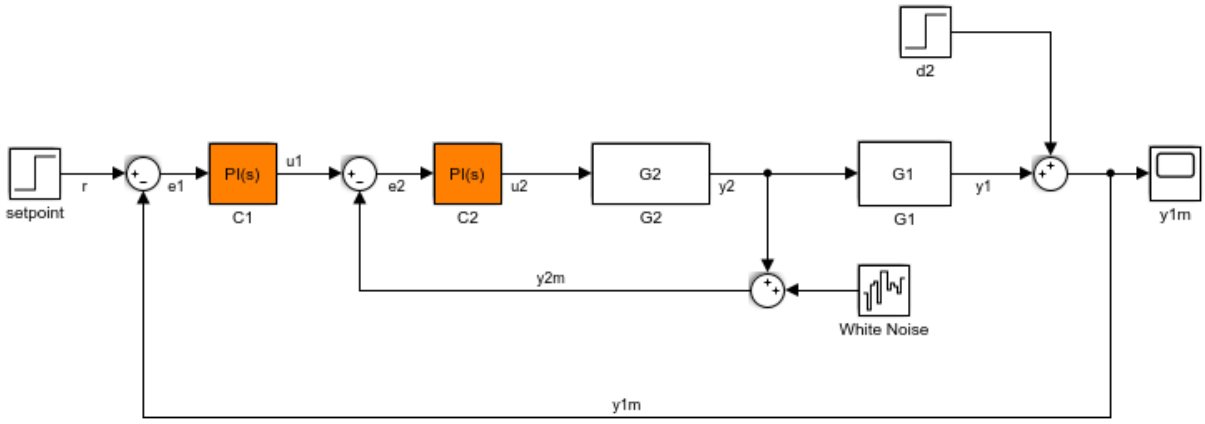
You can also use `setBlockParam` to set multiple block parameterizations at once, without requiring multiple recompilations of the model. For example, reparameterize both `C1` and `C2` as PID controllers.

```
C1PID = tunablePID('C1PID','PID');
C2PID = tunablePID('C2PID','PID');
setBlockParam(st,'C1',C1PID,'C2',C2PID);
```

Revert Parameterization of Tuned Block to Default

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Modify the parameterization of `C2` to be a tunable gain and examine the result.

```
G = tunableGain('C2',5);
setBlockParam(st,'C2',G);
getBlockParam(st,'C2')
```

```
ans =

  Parametric gain "C2" with 1 outputs, 1 inputs, and 1 tunable parameters.

Type "ss(ans)" to see the current value and "get(ans)" to see all properties.
```

Revert the parameterization of `C2` back to the default PI controller and examine the result.

```
setBlockParam(st,'C2');
getBlockParam(st,'C2')
```

```
ans =

  Parametric continuous-time PID controller "C2" with formula:

             1
  Kp + Ki * ---
             s
```

```
  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string | cell array of character vectors | string array

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

When reverting to the default block parameterization using `setBlockParam(st,blk)`, you can specify `blk` as a cell array of character vectors or string array to revert multiple blocks.

Example: `{'C1','C2'}`

`tunable_mdl` — Block parameterization

control design block | generalized state-space model | generalized matrix | tunable gain surface

Block parameterization, specified as one of the following:

- Control Design Block (Control System Toolbox)
- Generalized state-space (`genss`) model
- Generalized matrix (`genmat`)
- Tunable gain surface, modeled by `tunableSurface`

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`genss` | `getBlockParam` | `setBlockValue` | `setTunedValue` | `slTuner` | `systune` | `writeBlockValue`

Topics

"How Tuned Simulink Blocks Are Parameterized" on page 9-37

Introduced in R2011b

setBlockRateConversion

Set rate conversion settings for tuned block in `slTuner` interface

When you use `systune` with Simulink, tuning is performed at the sampling rate specified by the `Ts` property of the `slTuner` interface. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. The rate conversion method associated with each tuned block specifies how this resampling operation should be performed. Use `getBlockRateConversion` to query the block conversion rate and use `setBlockRateConversion` to modify it.

Syntax

```
setBlockRateConversion(st,blk,method)
setBlockRateConversion(st,blk,'tustin',pwf)

setBlockRateConversion(st,blk,IF,DF)
```

Description

`setBlockRateConversion(st,blk,method)` sets the rate conversion method of a tuned block on page 13-438 in the `slTuner` interface, `st`.

`setBlockRateConversion(st,blk,'tustin',pwf)` sets the Tustin method as the rate conversion method for `blk`, with `pwf` as the prewarp frequency.

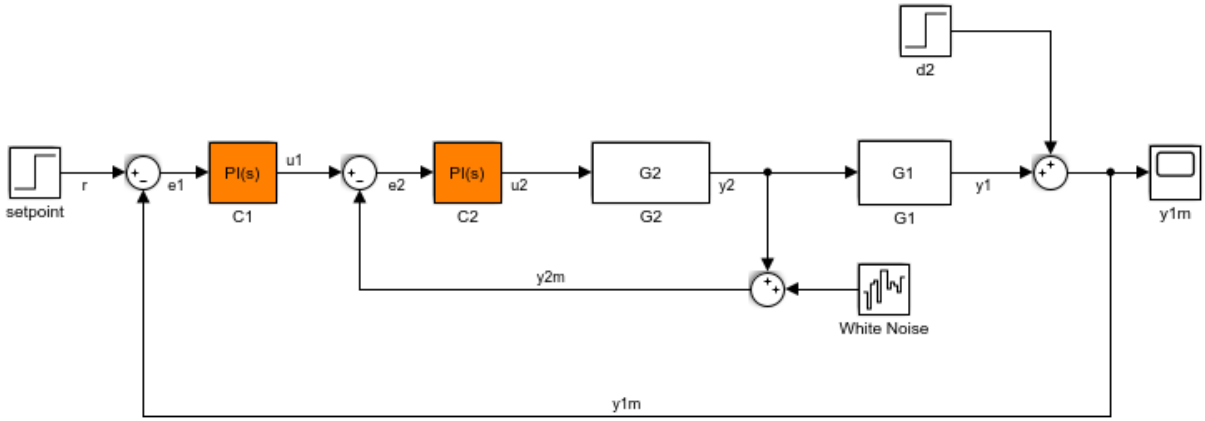
`setBlockRateConversion(st,blk,IF,DF)` sets the discretization methods for the integrator and derivative filter terms when `blk` is a continuous-time PID Controller block. For discrete-time PID blocks, these methods are specified in the Simulink block and cannot be modified in the `slTuner` interface.

Examples

Set Rate Conversion Settings of Tuned PID Block

Create an `slTuner` interface for the Simulink model `scdcascade`. Set the block rate conversion settings of one of the tuned blocks.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the default block rate conversion for the PID Controller block `C1`.

```
[IF,DF] = getBlockRateConversion(st,'C1')
```

```
IF =

    'Trapezoidal'

DF =

    'Trapezoidal'
```

By default, both the integrator and derivative filter controller methods are `Trapezoidal`. Set the integrator to `BackwardEuler` and the derivative to `ForwardEuler`.

```
IF = 'BackwardEuler';
DF = 'ForwardEuler';
setBlockRateConversion(st,'C1',IF,DF);
```

- "Tuning of a Digital Motion Control System" (Control System Toolbox)

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

`method` — Rate conversion method

`'zoh'` | `'foh'` | `'tustin'` | `'matched'`

Rate conversion method associated with `blk`, specified as one of the following:

- `'zoh'` — Zero-order hold on the inputs. This method is the default rate-conversion method for most dynamic blocks.
- `'foh'` — Linear interpolation of inputs.
- `'tustin'` — Bilinear (Tustin) approximation. Optionally, specify a prewarp frequency with the `pwf` argument for better frequency-domain matching between the original and rate-converted dynamics near the prewarp frequency.
- `'matched'` — Matched pole-zero method. This method is available for SISO blocks only.

For more detailed information about these rate-conversion methods, see "Continuous-Discrete Conversion Methods" (Control System Toolbox).

`pwf` — Prewarp frequency for Tustin method
positive scalar

Prewarp frequency for the Tustin method, specified as a positive scalar.

`IF,DF` — Integrator and filter methods
`'ForwardEuler' | 'BackwardEuler' | 'Trapezoidal'`

Integrator and filter methods for rate conversion of PID Controller block, each specified as one of the following:

- `'ForwardEuler'` — Integrator or derivative-filter state discretized as `Ts/(z-1)`
- `'BackwardEuler'` — `Ts*z/(z-1)`
- `'Trapezoidal'` — `(Ts/2)*(z+1)/(z-1)`

For continuous-time PID blocks, the default methods are `'Trapezoidal'` for both integrator and derivative filter. This method is the same as the Tustin method.

For discrete-time PID blocks, `IF` and `DF` are determined by the **Integrator method** and **Filter method** settings in the Simulink block and cannot be changed with `setBlockRateConversion`.

See the PID Controller and `pid` reference pages for more details about integrator and filter methods.

Definitions

Tuned Block

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tips

- For Model Discretizer blocks, the rate conversion method is specified in the Simulink block and cannot be modified with `setBlockRateConversion`.
- For static blocks such as Gain or Lookup Table blocks, the block rate conversion method is ignored.

See Also

`getBlockRateConversion` | `slTuner` | `writeBlockValue`

Topics

“Tuning of a Digital Motion Control System” (Control System Toolbox)
“Continuous-Discrete Conversion Methods” (Control System Toolbox)

Introduced in R2014a

setBlockValue

Set value of tuned block parameterization in `slTuner` interface

`setBlockValue` lets you initialize or modify the current value of the parameterization of a tuned block on page 13-445 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 13-446 for commands such as `systune`.

Syntax

```
setBlockValue(st,blk,value)

setBlockValue(st,blkValues)
```

Description

`setBlockValue(st,blk,value)` sets the current value of the parameterization of a block in the `slTuner` interface, `st`.

`setBlockValue(st,blkValues)` updates the values of the parameterizations of multiple blocks using the structure, `blkValues`.

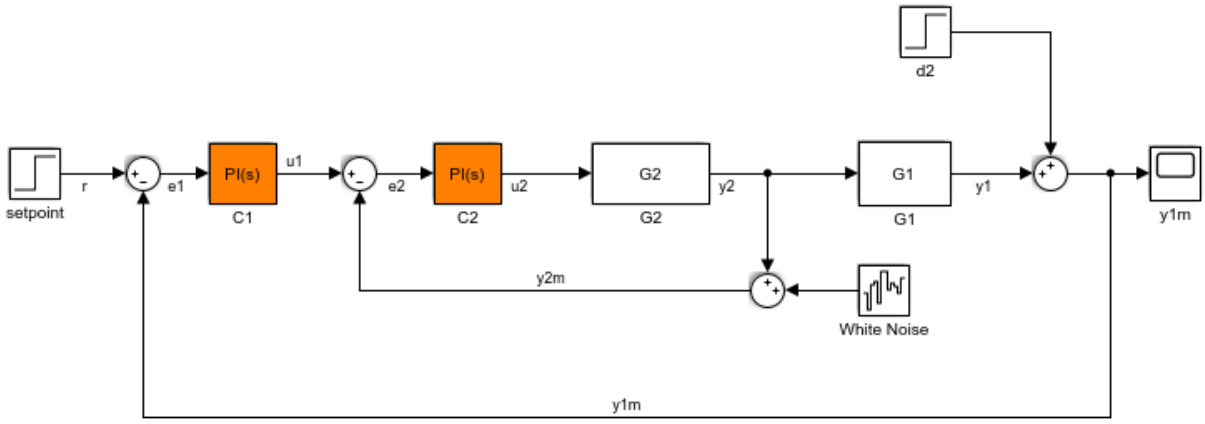
Examples

Set Value of Tuned Block Parameterization

Create an `slTuner` interface for the `scdcascade` model, and set the value of the parametrization of one of the tuned blocks.

Create an `slTuner` interface.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Both `C1` and `C2` are PI controllers. Examine the default parameterization of `C1`.

```
getBlockParam(st,'C1')
```

```
ans =

  Parametric continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

The default parameterization is a PI controller with two tunable parameters, `Kp` and `Ki`.

Set the value of the parameterization of `C1`.

```
C = pid(4.2);
setBlockValue(st,'C1',C);
```

Examine the value of the parameterization of `C1`.

```
getBlockValue(st,'C1')


ans =

  Kp = 4.2

Name: C1
P-only controller.
```

Examine the parameterization of C1.

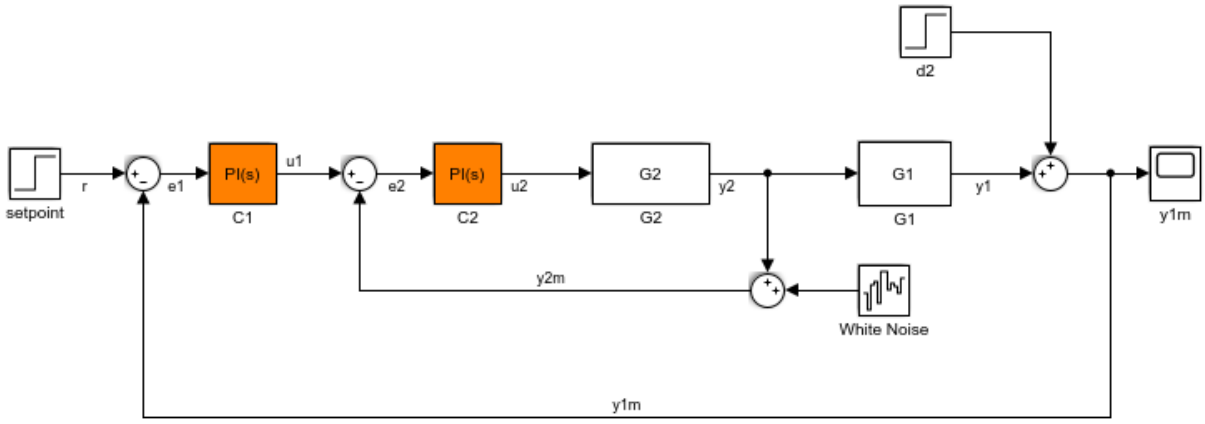
```
getBlockParam(st,'C1')


ans =

  Parametric continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

Observe that although the current block value is a P-only controller, the block parameterization continues to be a PI-controller.

Set Value of Multiple Tuned Block Parameterizations

Create an `slTuner` interface.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Create a block value structure with field names that correspond to the tunable blocks in `st`.

```
blockValues = getBlockValue(st);
blockValues.C1 = pid(0.2,0.1);
blockValues.C2 = pid(2.3);
```

Set the values of the parameterizations of the tunable blocks in `st` using the defined structure.

```
setBlockValue(st,blockValues);
```

- "Fixed-Structure Autopilot for a Passenger Jet" (Control System Toolbox)

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1', blk = "C1"`

Note `setBlockValue` allows you to modify only the overall value of the parameterization of `blk`. To modify the values of elements within custom block parameterizations, such as generalized state-space models, use `setTunedValue`.

`value` — Value of block parameterization

numeric LTI model | control design block

Value of block parameterization, specified as a numeric LTI model (Control System Toolbox) or a Control Design Block (Control System Toolbox), such `tunableGain` or `tunablePID`. The value of `value` must be compatible with the parameterization of `blk`. For example, if `blk` is parameterized as a PID controller, then `value` must be an `tunablePID` block, a numeric `pid` model, or a numeric `tf` model that represents a PID controller.

`setBlockValue` updates the value of the parameters of the tuned block based on the parameters of `value`. Using `setBlockValue` does not change the structure of the parameterization of the tuned block. To change the parameterization of `blk`, use `setBlockParam`. For example, you can use `setBlockParam` to change a block parameterization from `tunablePID` to a three-pole `tunableTF` model.

`blkValues` — Values of multiple block parameterizations

structure

Values of multiple block parameterizations, specified as a structure with fields specified as numeric LTI models or Control Design Blocks. The field names are the names of blocks in `st`. Only blocks common to `st` and `blkValues` are updated, while all other blocks in `st` remain unchanged.

To specify `blkValues`, you can retrieve and modify the block parameterization value structure from `st`.

```
blkValues = getblockValue(st);
blkValues.C1 = pid(0.1,0.2);
```

Note For Simulink blocks whose names are not valid field names, specify the corresponding field name in `blkValues` as it appears in the block parameterization.

```
blockParam = getBlockParam(st,'B-1');
fieldName = blockParam.Name;
blockValues = struct(fieldName,newB1);
```

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`getBlockValue` | `setBlockParam` | `setTunedValue` | `slTuner` | `writeBlockValue`

Topics

“Fixed-Structure Autopilot for a Passenger Jet” (Control System Toolbox)
“How Tuned Simulink Blocks Are Parameterized” (Control System Toolbox)

Introduced in R2011b

setTunedValue

Set current value of tuned variable in `slTuner` interface

`setTunedValue` lets you initialize or modify the current value of a tuned variable on page 13-453 within an `slTuner` interface.

An `slTuner` interface parameterizes each tuned block on page 13-453 as a Control Design Block (Control System Toolbox), or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables for commands such as `systune`.

Syntax

```
setTunedValue(st,var,value)
setTunedValue(st,varValues)
setTunedValue(st,model)
```

Description

`setTunedValue(st,var,value)` sets the current value of the tuned variable, `var`, in the `slTuner` interface, `st`.

`setTunedValue(st,varValues)` sets the values of multiple tuned variables in `st` using the structure, `varValues`.

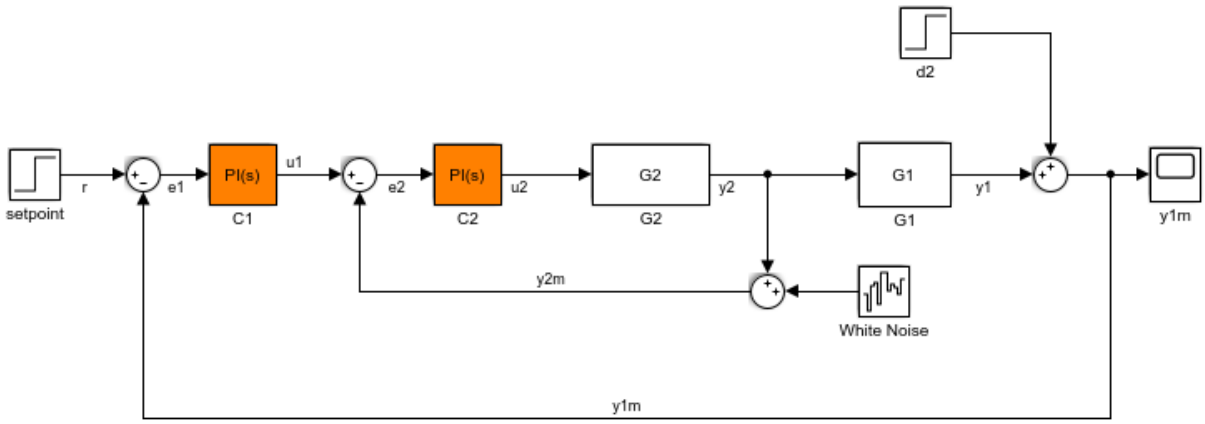
`setTunedValue(st,model)` updates the values of the tuned variables in `st` to match their values in the generalized model `model`. To propagate tuned values from one model to another, use this syntax.

Examples

Set Value of Single Tunable Element within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

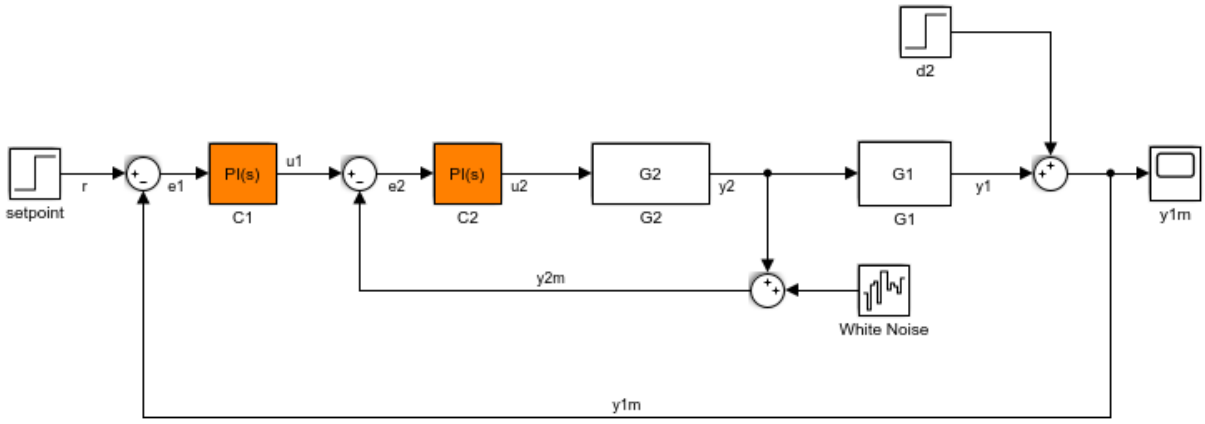
Initialize the value of `Ki` to `10` without changing the value of Kp.

```
setTunedValue(st,'Ki',10);
```

Set Value of Multiple Tunable Elements within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

Create a structure of tunable element values, setting `Kp` to `5` and `Ki` to `10`.

```
S = struct('Kp',5,'Ki',10);
```

Set the values of the tunable elements in `st`.

```
setTunedValue(st,S);
```

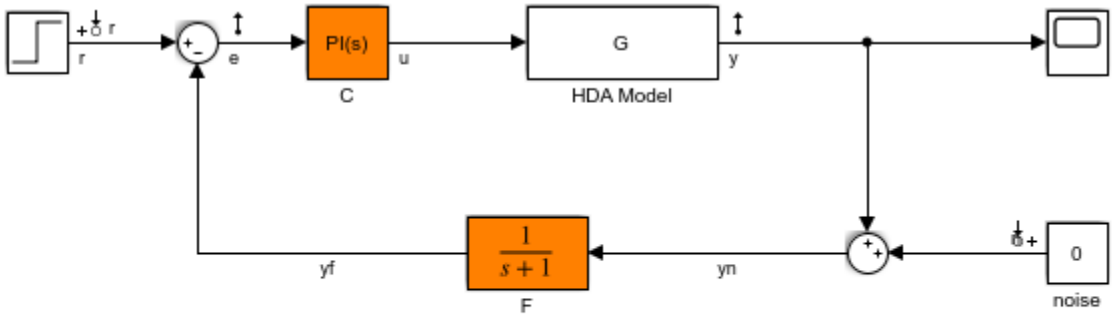
Set Value of Tuned Block Parameterization Using Generalized State-Space Model

Convert an `slTuner` interface for the Simulink® model `rct_diskdrive` to a `genss` model to tune the model blocks using `hinfstruct`. After tuning, update the `slTuner` interface with the tuned parameters and write the parameter values to the Simulink model for validation.

Use of `hinfstruct` requires a Robust Control Toolbox license.

Create an `slTuner` interface for `rct_diskdrive`. Add `C` and `F` as tuned blocks of the interface.

```
open_system('rct_diskdrive');
st = slTuner('rct_diskdrive',{'C','F'});
```



See hinfstruct_demo to see how you can tune the PI gains and the filter coefficient with the HINFSTRUCT command.

The default parameterization of the transfer function block, `F`, is a transfer function with two free parameters. Because `F` is a low-pass filter, you must constrain its coefficients. To do so, specify a custom parameterization of `F` with filter coefficient `a`.

```
a = realp('a',1);
setBlockParam(st,'F',tf(a,[1 a]));
```

Convert `st` to a `genss` model.

```
m = getIOTransfer(st,{'r','n'},{'y','e'});
```

Typically, for tuning with `hinfstruct`, you append weighting functions to the `genss` model that depend on your design requirements. You then tune the augmented model. For more information, see “Fixed-Structure H-infinity Synthesis with HINFSTRUCT” (Robust Control Toolbox).

For this example, instead of tuning the model, manually adjust the tuned variable values.

```
m.Blocks.C.Kp.Value = 0.00085;
m.Blocks.C.Ki.Value = 0.01;
m.Blocks.a.Value = 5500;
```

After tuning, update the block parameterization values in `st`.

```
setTunedValue(st,m);
```

This is equivalent to `setBlockValue(st,m.Blocks)`.

To validate the tuning result in Simulink, first update the Simulink model with the tuned values.

```
writeBlockValue(st);
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`var` — Tuned variable

character vector | string

Tuned variable within `st`, specified as a character vector or string. A tuned variable is any Control Design Block, such `realp`, `tunableSS`, or `tunableGain`, involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. To get a list of all tuned variables within `st`, use `getTunedValue(st)`.

`var` can refer to the following:

- For a block parameterized by a Control Design Block, the name of the block. For example, if the parameterization of the block is

  ```
  C = tunableSS('C')
  ```

  then set `var = 'C'`.

- For a block parameterized by a `genmat`/`genss` model, `M`, the name of any Control Design Block listed in `M.Blocks`. For example, if the parameterization of the block is

  ```
  a = realp('a',1);
  C = tf(a,[1 a]);
  ```

  then set `var` = `'a'`.

`value` — Value of tuned variable

numeric scalar | numeric array | state-space model

Value of tuned variable in `st`, specified as a numeric scalar, a numeric array or a state-space model that is compatible with the tuned variable. For example, if `var` is a scalar element such as a PID gain, `value` must be a scalar. If `var` is a 2–by–2 `tunableGain`, then `value` must be a 2–by–2 scalar array.

`varValues` — Values of multiple tuned variables

structure

Values of multiple tuned variables in `st`, specified as a structure with fields specified as numeric scalars, numeric arrays, or state-space models. The field names are the names of tuned variables in `st`. Only blocks common to `st` and `varValues` are updated, while all other blocks in `st` remain unchanged.

To specify `varValues`, you can retrieve and modify the tuned variable structure from `st`.

```
varValues = getTunedValue(st);
varValues.Ki = 10;
```

`model` — Tuned model

generalized LTI model

Tuned model that has some parameters in common with `st`, specified as a Generalized LTI Model (Control System Toolbox). Only variables common to `st` and `model` are updated, while all other variables in `st` remain unchanged.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks (Control System Toolbox) involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`getTunedValue` | `setBlockParam` | `setBlockValue` | `slTuner` | `tunableSurface` | `writeBlockValue`

Topics

"How Tuned Simulink Blocks Are Parameterized" (Control System Toolbox)

Introduced in R2015b

showTunable

Show value of parameterizations of tunable blocks of `slTuner` interface

Syntax

```
showTunable(st)
```

Description

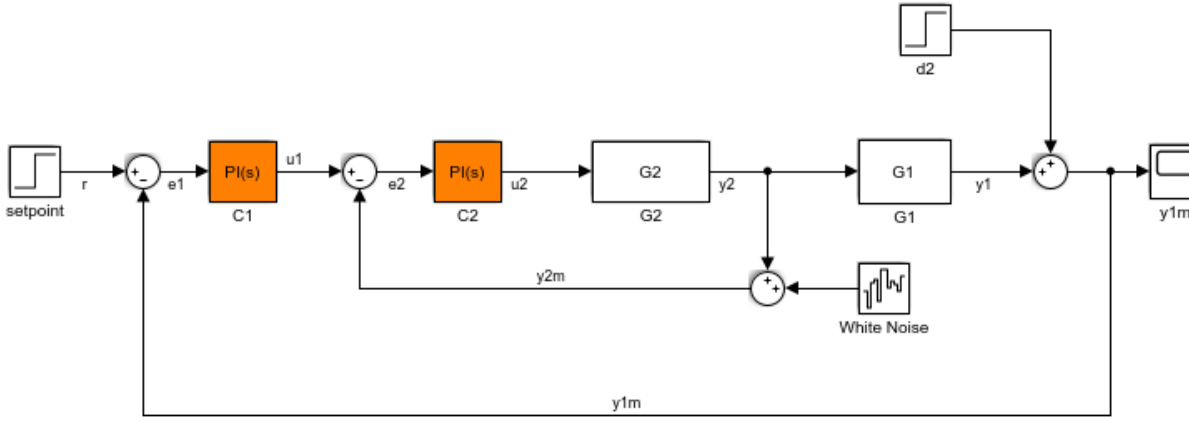
`showTunable(st)` displays the values of the parameteric models associated with each tunable block on page 13-457 in the `slTuner` interface, `st`.

Examples

Display Tunable Block Values

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slTuner` interface for the model, and add `C1` and `C2` as tuned blocks of the interface.

```
st = slTuner(mdl,{'C1','C2'});
```

Display the default values of the tuned blocks.

```
showTunable(st);
```

```
Block 1: scdcascade/C1 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.158, Ki = 0.042

Name: C1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: scdcascade/C2 =

             1
  Kp + Ki * ---
             s
```

```
  with Kp = 1.48, Ki = 4.76

Name: C2
Continuous-time PI controller in parallel form.
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`getBlockValue` | `setBlockValue` | `slTuner` | `writeBlockValue`

Introduced in R2014a

systune

Tune control system parameters in Simulink using `slTuner` interface

`systune` tunes fixed-structure control systems subject to both soft and hard design goals. `systune` can tune multiple fixed-order, fixed-structure control elements distributed over one or more feedback loops. For an overview of the tuning workflow, see "Automated Tuning Workflow" (Control System Toolbox) in the Control System Toolbox documentation.

This command tunes control systems modeled in Simulink. For tuning control systems represented in MATLAB, `systune` for `genss` models.

Syntax

```
[st,fSoft] = systune(st0,SoftGoals)
[st,fSoft,gHard] = systune(st0,SoftGoals,HardGoals)
[st,fSoft,gHard] = systune(___,opt)
[st,fSoft,gHard,info] = systune(___)
```

Description

`[st,fSoft] = systune(st0,SoftGoals)` tunes the free parameters of the control system in Simulink. The Simulink model, tuned blocks on page 13-468, and analysis points on page 13-468 of interest are specified by the `slTuner` interface, `st0`. `systune` tunes the control system parameters to best meet the performance goals, `SoftGoals`. The command returns a tuned version of `st0` as `st`. The best achieved soft constraint values are returned as `fSoft`.

If the `st0` contains real parameter uncertainty, `systune` automatically performs robust tuning to optimize the constraint values for worst-case parameter values. `systune` also performs robust tuning against a set of plant models obtained at different operating points or parameter values. See "Input Arguments" on page 13-463.

Tuning is performed at the sample time specified by the `Ts` property of `st0`.

`[st,fSoft,gHard] = systune(st0,SoftGoals,HardGoals)` tunes the control system to best meet the soft goals, subject to satisfying the hard goals. It returns the best achieved values, `fSoft` and `gHard`, for the soft and hard goals. A goal is met when its achieved value is less than 1.

`[st,fSoft,gHard] = systune(___,opt)` specifies options for the optimization for any of the input argument combinations in previous syntaxes.

`[st,fSoft,gHard,info] = systune(___)` also returns detailed information about each optimization run for any of the input argument combinations in previous syntaxes.

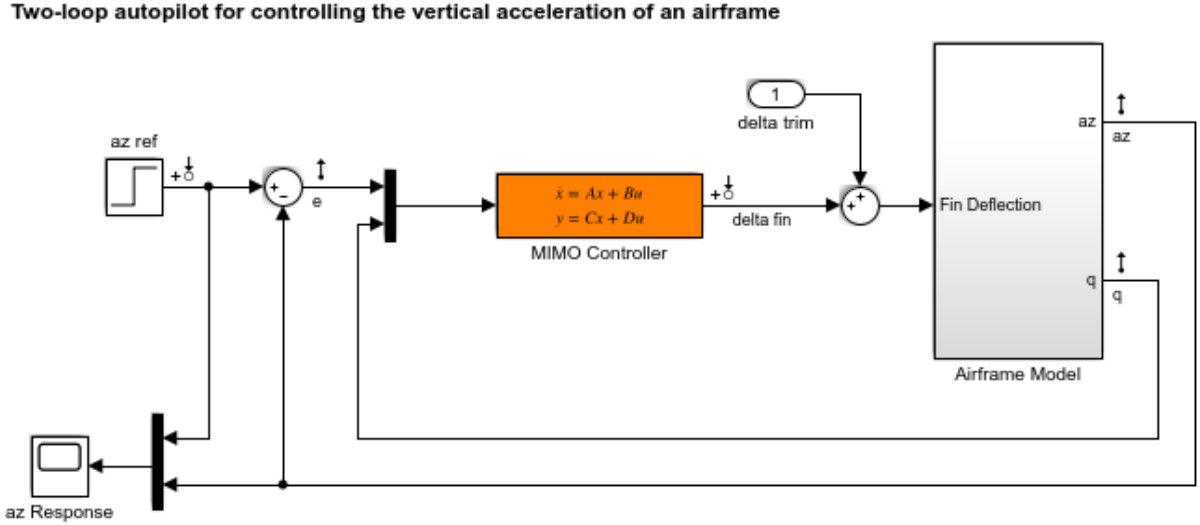
Examples

Tune Control System to Soft Constraints

Tune the control system in the `rct_airframe2` model to soft goals for tracking, roll off, stability margin, and disturbance rejection.

Open the Simulink model.

```
mdl = 'rct_airframe2';
open_system(mdl);
```



Create and configure an `slTuner` interface to the model.

```
st0 = slTuner(mdl,'MIMO Controller');
```

`st0` is an `slTuner` interface to the `rct_aircraft2` model with the `MIMO Controller` block specified as the tunable portion of the control system.

The model already has linearization input points on the signals `az ref`, `delta fin`, `az`, `q`, and `e`. These signals are therefore available as analysis points for tuning goals and linearization.

Specify the tracking requirement, roll-off requirement, stability margins, and disturbance rejection requirement.

```
req1 = TuningGoal.Tracking('az ref','az',1);
req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
req3 = TuningGoal.Margins('delta fin',7,45);
max_gain = frd([2 200 200],[0.02 2 200]);
req4 = TuningGoal.Gain('delta fin','az',max_gain);
```

`req1` constrains `az` to track `az ref`. The next requirement, `req2`, imposes a roll-off requirement by specifying a gain profile for the open-loop, point-to-point transfer function measured at `delta fin`. The next requirement, `req3`, imposes open-loop gain

and phase margins on that same point-to-point transfer function. Finally, `req4` rejects disturbances to `az` injected at `delta fin`, by specifying a maximum gain profile between those two points.

Tune the model using these tuning goals.

```
opt = systuneOptions('RandomStart',3);
rng(0);
[st,fSoft,~,info] = systune(st0,[req1,req2,req3,req4],opt);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 69
Final: Soft = 1.13, Hard = -Inf, Iterations = 115
Final: Soft = 1.13, Hard = -Inf, Iterations = 102
Final: Failed to enforce closed-loop stability (max Re(s) = -0)
```

`st` is a tuned version of `st0`.

The `RandomStart` option specifies that `systune` must perform three independent optimization runs that use different (random) initial values of the tunable parameters. These three runs are in addition to the default optimization run that uses the current value of the tunable parameters as the initial value. The call to `rng` seeds the random number generator to produce a repeatable sequence of numbers.

`systune` displays the final result for each run. The displayed value, `Soft`, is the maximum of the values achieved for each of the four performance goals. The software chooses the best run overall, which is the run yielding the lowest value of `Soft`. The last run fails to achieve closed-loop stability, which corresponds to `Soft = Inf`.

Examine the best achieved values of the soft constraints.

```
fSoft
```

```
fSoft =

    1.1327    1.1327    0.5140    1.1327
```

Only `req3`, the stability margin requirement, is met for all frequencies. The other values are close to, but exceed, 1, indicating violations of the goals for at least some frequencies.

Use `viewGoal` to visualize the tuned control system performance against the goals and to determine whether the violations are acceptable. To evaluate specific open-loop or

closed-loop transfer functions for the tuned parameter values, you can use linearization commands such as `getIOTransfer` and `getLoopTransfer`. After validating the tuned parameter values, if you want to apply these values to the Simulink® model, you can use `writeBlockValue`.

- "Tune Control Systems in Simulink" (Control System Toolbox)
- "Control of a Linear Electric Actuator" (Control System Toolbox)
- "Interpret Numeric Tuning Results" on page 9-183

Input Arguments

`st0` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

If you specify parameter variation or linearization at multiple operating points when you create `st0`, then `systune` performs robust tuning against all the plant models. If you specify an uncertain (`uss`) model as a block substitution when you create `st0`, then `systune` performs robust tuning, optimizing the parameters against the worst-case parameter values. For more information about robust tuning approaches, see "Robust Tuning Approaches" (Robust Control Toolbox). (Using uncertain models requires a Robust Control Toolbox license.)

`SoftGoals` — Soft goals (objectives)

vector of `TuningGoal` objects

Soft goals (objectives) for tuning the control system described by `st0`, specified as a vector of `TuningGoal` objects. For a complete list, see "Tuning Goals".

`systune` tunes the tunable parameters of the control system to minimize the maximum value of the soft tuning goals, subject to satisfying the hard tuning goals (if any).

`HardGoals` — Hard goals (constraints)

vector of `TuningGoal` objects

Hard goals (constraints) for tuning the control system described by `st0`, specified as a vector of `TuningGoal` objects. For a complete list, see "Tuning Goals".

A hard goal is satisfied when its value is less than 1. `systune` tunes the tunable parameters of the control system to minimize the maximum value of the soft tuning goals, subject to satisfying all the hard tuning goals.

`opt` — Tuning algorithm options
options set created using `systuneOptions`

Tuning algorithm options, specified as an options set created using `systuneOptions`.

Available options include:

- Number of additional optimizations to run starting from random initial values of the free parameters
- Tolerance for terminating the optimization
- Flag for using parallel processing

See the `systuneOptions` reference page for more details about all available options.

Output Arguments

`st` — Tuned interface
`slTuner` interface

Tuned interface, returned as an `slTuner` interface.

`fSoft` — Best achieved values of soft goals
vector

Best achieved values of soft goals, returned as a vector.

Each tuning goal evaluates to a scalar value, and `systune` minimizes the maximum value of the soft goals, subject to satisfying all the hard goals.

`fSoft` contains the value of each soft goal for the best overall run. The best overall run is the run that achieved the smallest value for `max(fSoft)`, subject to `max(gHard)<1`.

`gHard` — Achieved values of hard goals
vector

Achieved values of hard goals, returned as a vector.

`gHard` contains the value of each hard goal for the best overall run (the run that achieved the smallest value for `max(fSoft)`, subject to `max(gHard)<1`. All entries of `gHard` are less than 1 when all hard goals are satisfied. Entries greater than 1 indicate that `systune` could not satisfy one or more design constraints.

`info` — Detailed information about each optimization run
structure

Detailed information about each optimization run, returned as a structure. The fields of `info` are:

`Run` — Run number
scalar

Run number, returned as a scalar. If you use the `RandomStart` option of `systuneOptions` to perform multiple optimization runs, `info` is a struct array, and `info.Run` is the index.

`Iterations` — Total number of iterations performed during run
scalar

Total number of iterations performed during run, returned as a scalar.

`fBest` — Best overall soft constraint value
scalar

Best overall soft constraint value, returned as a scalar. `systune` converts the soft goals to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard constraints. (See “Algorithms” on page 13-469.) `info.fBest` is the maximum soft constraint value at the final iteration. This value is only meaningful when the hard constraints are satisfied.

`gBest` — Best overall hard constraint value
scalar

Best overall hard constraint value, returned as a scalar. `systune` converts the hard goals to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See “Algorithms” on page 13-469.) `info.gBest` is the maximum hard constraint value at the final iteration. This value must be less than 1 for the hard constraints to be satisfied.

`fSoft` — Individual soft constraint values

vector

Individual soft constraint values, returned as a vector. `systune` converts each soft requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize that value subject to the hard constraints. (See “Algorithms” on page 13-469.) `info.fSoft` contains the individual values of the soft constraints at the end of each run. These values appear in `fSoft` in the same order that the constraints are specified in `SoftGoals`.

`gHard` — Individual hard constraint values

vector

Individual hard constraint values, returned as a vector. `systune` converts each hard requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize those values. A hard requirement is satisfied if its value is less than 1. (See “Algorithms” on page 13-469.) `info.gHard` contains the individual values of the hard constraints at the end of each run. These values appear in `gHard` in the same order that the constraints are specified in `HardGoals`.

`MinDecay` — Minimum decay rate of closed-loop poles

vector

Minimum decay rate of closed-loop poles, returned as a vector.

By default, closed-loop pole locations of the tuned system are constrained to satisfy $\mathrm{Re}(p) < -10^{-7}$. Use the `MinDecay` option of `systuneOptions` to change this constraint.

`Blocks` — Tuned values of tunable blocks and parameters

structure

Tuned values of tunable blocks and parameters, returned as a structure.

In case of multiple runs, you can try the results of any particular run other than the best run. To do so, you can use either `getBlockValue` or `showTunable` to access the tuned parameter values. For example, to use the results from the third run, type `getBlockValue(st,Info(3).Blocks)`.

`LoopScaling` — Optimal diagonal scaling for evaluating MIMO tuning goals

state-space model

Optimal diagonal scaling for evaluating MIMO tuning goals, returned as a state-space model.

When applied to multiloop control systems, tuning goals such as `TuningGoal.LoopShape` and `TuningGoal.Margins` can be sensitive to the scaling of the individual loop transfer functions to which they apply. `systune` automatically corrects scaling issues and returns the optimal diagonal scaling matrix `d` as a state-space model in `info.LoopScaling`.

The loop channels associated with each diagonal entry of `D` are listed in `info.LoopScaling.InputName`. The scaled loop transfer is `D\L*D`, where `L` is the open-loop transfer measured at the locations `info.LoopScaling.InputName`.

`wcPert` — Worst combinations of uncertain parameters
structure array

Worst combinations of uncertain parameters, returned as a structure array. (Applies for robust tuning of control systems with uncertainty only.) Each structure contains one set of uncertain parameter values. The perturbations with the worst performance are listed first.

`wcf` — Worst objective value
positive scalar

Largest soft goal value over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.)

`wcg` — Worst constraint value
positive scalar

Largest hard goal value over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.)

`wcDecay` — Worst decay rate
scalar

Smallest closed-loop decay rate over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.) A positive value indicates robust stability. See `MinDecay` option in `systuneOptions` for details.

Definitions

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see "How Tuned Simulink Blocks Are Parameterized" on page 9-37). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model (Control System Toolbox).

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see "Mark Signals of Interest for Control System Analysis and Design" on page 2-48 and "Mark Signals of Interest for Batch Linearization" on page 3-13.

Algorithms

x is the vector of tunable parameters in the control system to tune. `systune` converts each soft and hard tuning requirement `SoftReqs(i)` and `HardReqs(j)` into normalized values $f_i(x)$ and $g_j(x)$, respectively. `systune` then solves the constrained minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system.

When you use both soft and hard tuning goals, the software approaches this optimization problem by solving a sequence of unconstrained subproblems of the form:

$$\min_x \ \max\left(\alpha f(x), g(x)\right).$$

The software adjusts the multiplier α so that the solution of the subproblems converges to the solution of the original constrained optimization problem.

`systune` returns the `slTuner` interface with parameters tuned to the values that best solve the minimization problem. `systune` also returns the best achieved values of $f_i(x)$ and $g_j(x)$, as `fSoft` and `gHard` respectively.

For information about the functions $f_i(x)$ and $g_j(x)$ for each type of constraint, see the reference pages for each `TuningGoal` requirement object.

`systune` uses the nonsmooth optimization algorithms described in [1],[2],[3],[4]

`systune` computes the H_∞ norm using the algorithm of [5] and structure-preserving eigensolvers from the SLICOT library. For information about the SLICOT library, see http://slicot.org.

Alternative Functionality

Tune interactively using **Control System Tuner**.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.

[2] Apkarian, P. and D. Noll, "Nonsmooth Optimization for Multiband Frequency-Domain Control Design," *Automatica*, 43 (2007), pp. 724–731.

[3] Apkarian, P., P. Gahinet, and C. Buhr, "Multi-model, multi-objective tuning of fixed-structure controllers," *Proceedings ECC* (2014), pp. 856–861.

[4] Apkarian, P., M.-N. Dao, and D. Noll, "Parametric Robust Structured Control Design," *IEEE Transactions on Automatic Control*, 2015.

[5] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

See Also

`addPoint` | `getIOTransfer` | `getLoopTransfer` | `hinfstruct` | `looptune` | `slTuner` | `systune (for genss)` | `systuneOptions` | `writeBlockValue`

Topics

"Tune Control Systems in Simulink" (Control System Toolbox)
"Control of a Linear Electric Actuator" (Control System Toolbox)
"Interpret Numeric Tuning Results" on page 9-183
"Tuning Goals"
"Robust Tuning Approaches" (Robust Control Toolbox)

Introduced in R2014a

writeBlockValue

Update block values in Simulink model

Syntax

```
writeBlockValue(st)
writeBlockValue(st,blockid)
writeBlockValue(st,m)
```

Description

`writeBlockValue(st)` writes tuned parameter values from the `slTuner` interface, `st`, to the Simulink model that `st` describes. Use this command, for example, to validate parameters of a control system that you tuned using `systune` or `looptune`.

`writeBlockValue` skips blocks that cannot represent their tuned value in a straightforward and lossless manner. For example, suppose you tune an user defined Subsystem or S-Function block. `writeBlockValue` will skip this block because there is no clear way to map the tuned value to a Subsystem or S-Function block. Similarly, if you parameterize a Gain block as a second-order transfer function, `writeBlockValue` will skip this block, unless the transfer function value is a static gain.

`writeBlockValue(st,blockid)` only updates the block or blocks referenced by `blockid`.

`writeBlockValue(st,m)` writes tuned parameter values from a generalized model, `m`, to the Simulink model described by the `slTuner` interface, `st`.

Examples

Update Simulink Model with All Tuned Parameters

Create an `slTuner` interface for the model.

```
st = slTuner('scdcascade',{'C1','C2'});
```

Specify the tuning goals and necessary analysis points.

```
tg1 = TuningGoal.StepTracking('r','y1m',5);

addPoint(st,{'r','y1m'});

tg2 = TuningGoal.Poles();
tg2.MaxFrequency = 10;
```

Tune the controller.

```
[sttuned,fSoft] = systune(st,[tg1 tg2]);

Final: Soft = 1.28, Hard = -Inf, Iterations = 37
```

After validating the tuning results, update the model to use the tuned controller values.

```
writeBlockValue(sttuned);
```

- “Tuning of a Digital Motion Control System” (Control System Toolbox)
- “Control of a Linear Electric Actuator” (Control System Toolbox)

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blockid` — Blocks to update

character vector | string | cell array of character vectors | string array

Blocks to update with tuned values, specified as a:

- Character vector or string, to update one block.
- Cell array of character vectors or string array, to update multiple blocks.

The blocks in `blockid` must be in the `TunedBlocks` property of the `slTuner` interface `st`. You can specify a full block path, or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = {'scdcascade/C1','scdcascade/C2'}`

Example: `"C1"`

`m` — Tuned control system

generalized state-space

Tuned control system, specified as a generalized state-space model (`genss`).

Typically, `m` is the output of a tuning function like `systune`, `looptune`, or `hinfstruct`. The model `m` must have some tunable parameters in common with `st`. For example, `m` can be a generalized model that you obtained by linearizing your Simulink model, and then tuned to meet some design requirements.

See Also

`getBlockValue` | `setBlockValue` | `showTunable` | `slTuner` | `writeLookupTableData`

Topics

“Tuning of a Digital Motion Control System” (Control System Toolbox)
“Control of a Linear Electric Actuator” (Control System Toolbox)
“How Tuned Simulink Blocks Are Parameterized” (Control System Toolbox)

Introduced in R2014a

slTuner

Interface for control system tuning of Simulink models

Syntax

```
st = slTuner(mdl,tuned_blocks)
st = slTuner(mdl,tuned_blocks,pt)
st = slTuner(mdl,tuned_blocks,param)
st = slTuner(mdl,tuned_blocks,op)
st = slTuner(mdl,tuned_blocks,blocksub)
st = slTuner(mdl,tuned_blocks,options)
st = slTuner(mdl,tuned_blocks,pt,op,param,blocksub,options)
```

Description

`st = slTuner(mdl,tuned_blocks)` creates an `slTuner` interface, `st`, for tuning the control system blocks of the Simulink model, `mdl`. The interface adds the linear analysis points marked in the model as analysis points on page 13-489 of `st`. The interface also adds the linear analysis points that imply an opening as permanent openings on page 13-490. When the interface performs linearization, for example, to tune the blocks, it uses the model initial condition as the operating point.

`st = slTuner(mdl,tuned_blocks,pt)` adds the specified point to the list of analysis points for `st`, ignoring linear analysis points marked in the model.

`st = slTuner(mdl,tuned_blocks,param)` specifies the parameters whose values you want to vary when tuning the model blocks.

`st = slTuner(mdl,tuned_blocks,op)` specifies the operating points for tuning the model blocks.

`st = slTuner(mdl,tuned_blocks,blocksub)` specifies substitute linearizations of blocks and subsystems. Use this syntax, for example, to specify a custom linearization for a block. You can also use this syntax for blocks that do not linearize successfully, such as blocks with discontinuities or triggered subsystems.

`st = slTuner(mdl,tuned_blocks,options)` configures the linearization algorithm options.

`st = slTuner(mdl,tuned_blocks,pt,op,param,blocksub,options)` uses any combination of the input arguments `pt`, `op`, `param`, `blocksub`, and `options` to create `st`. For example, you can use:

- `st = slTuner(mdl,tuned_blocks,pt,param)`
- `st = slTuner(mdl,tuned_blocks,op,param)`.

Object Description

`slTuner` provides an interface between a Simulink model and the tuning commands `systune` and `looptune`. `slTuner` allows you to:

- Specify the control architecture.
- Designate and parameterize blocks to be tuned.
- Tune the control system.
- Validate design by computing (linearized) open-loop and closed-loop responses.
- Write tuned values back to the model.

Because tuning commands such as `systune` operate on linear models, the `slTuner` interface automatically computes and stores a linearization of your Simulink model. This linearization is automatically updated when you change any properties of the `slTuner` interface. The update occurs when you call commands that query the linearization stored in the interface. Such commands include `systune`, `looptune`, `getIOTransfer`, and `getLoopTransfer`. For more information about linearization, see “What Is Linearization?” on page 2-3

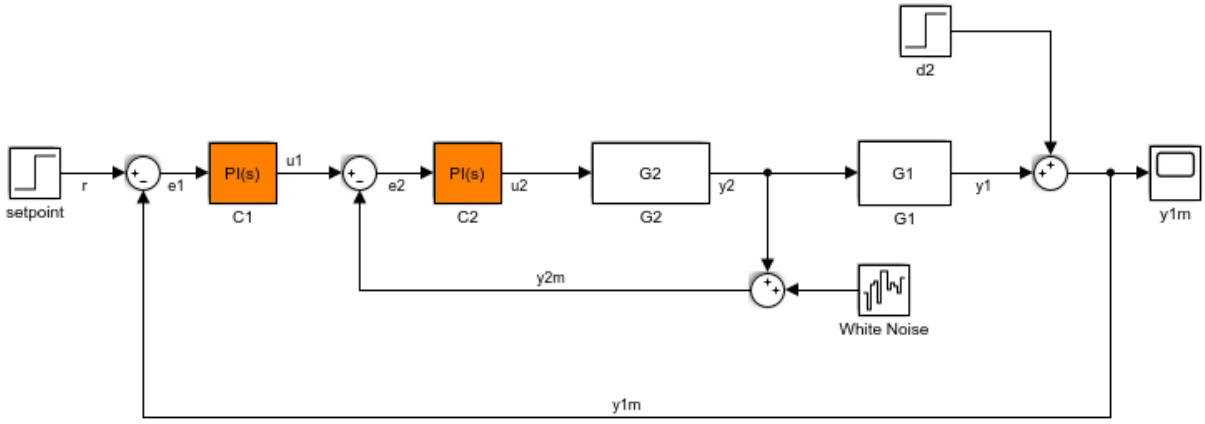
Examples

Create and Configure slTuner Interface for Control System Tuning

Create and configure an `slTuner` interface for a Simulink® model that specifies which blocks to tune with `systune` or `looptune`.

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl);
```



The control system consists of two feedback loops, an inner loop with PI controller `C2`, and an outer loop with PI controller `C1`. Suppose you want to tune this model to meet the following control objectives:

- Track setpoint changes to `r` at the system output `y1m` with zero steady-state error and specified rise time.
- Reject the disturbance represented by `d2`.

The `systune` command can jointly tune the controller blocks to meet these design requirements, which you specify using `TuningGoal` objects. The `slTuner` interface sets up this tuning task.

Create an `slTuner` interface for the model.

```
st = slTuner(mdl,{'C1','C2'});
```

This command initializes the `slTuner` interface and designates the two PI controller blocks as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model. A linearization of the remaining, nontunable portion of the model is computed and stored in the `slTuner` interface.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. Add the output and reference input for the tracking requirement. Also, add the disturbance-rejection location.

```
addPoint(st,{'r','y1m','d2'});
```

These locations in your model are now available for referencing in `TuningGoal` objects that capture your design goals.

Display a summary of the `slTuner` interface configuration in the command window.

```
st
```

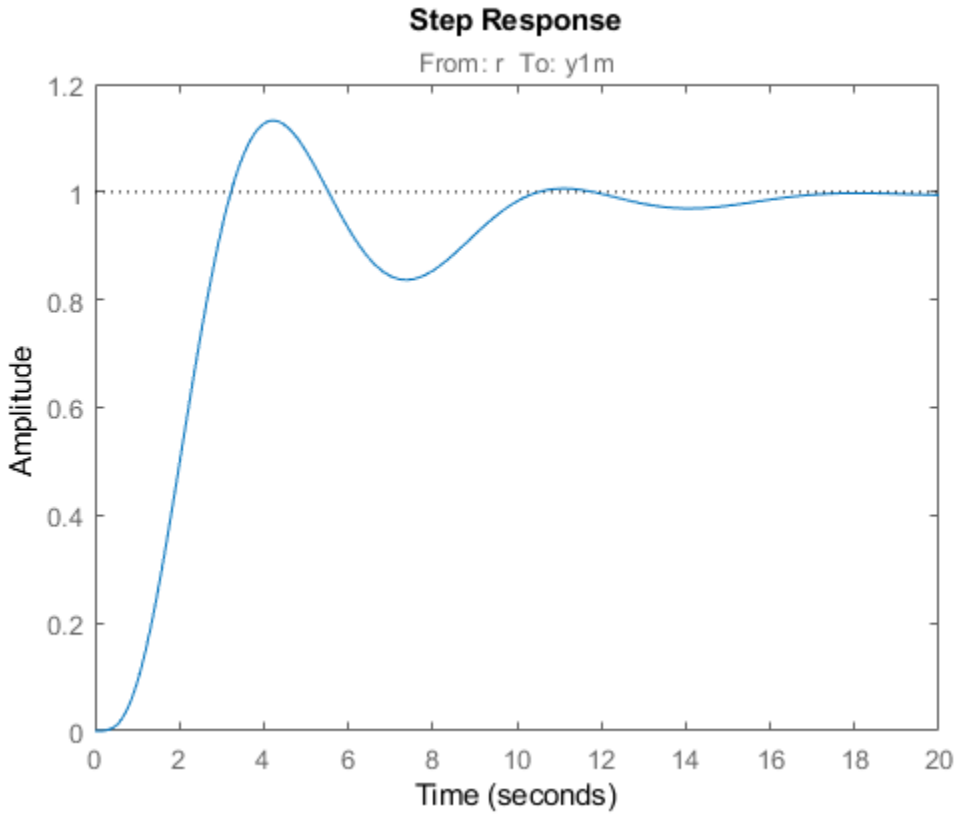
```
slTuner tuning interface for "scdcascade":

2 Tuned blocks: (Read-only TunedBlocks property)
---------------------------
Block 1: scdcascade/C1
Block 2: scdcascade/C2

3 Analysis points:
---------------------------
Point 1: Signal "r", located at port 1 of scdcascade/setpoint
Point 2: Signal "y1m", located at port 1 of scdcascade/Sum
Point 3: Port 1 of scdcascade/d2

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

The display lists the designated tunable blocks, analysis points, and other information about the interface. In the command window, click on any highlighted signal to see its location in the Simulink model. Note that specifying the block name `'d2'` in the `addPoint` command is equivalent to designating that block's single output signal as the analysis point.

You can now capture your design goals with `TuningGoal` objects and use `systune` or `looptune` to tune the control system to meet those design goals.

In addition to specifying design goals, you can use analysis points for extracting system responses. For example, extract and plot the step response between the reference signal `'r'` and the output `'y1m'`.

```
T = getIOTransfer(st,'r','y1m');
stepplot(T)
```



Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string.

Example: `'scdcascade'`

`tuned_blocks` — Blocks to be tuned

character vector | string | cell array of character vectors | string array

Blocks to be added to the list of tuned blocks of `st`, specified as:

- Character vector or string — Block path. You can specify the full block path or a partial path. The partial path must match the end of the full block path and unambiguously identify the block to add. For example, you can refer to a block by its name, provided the block name appears only once in the Simulink model.

  For example, `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array — Multiple block paths.

  For example, `blk = {'scdcascade/C1','scdcascade/C2'}`.

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point on page 13-489 to be added to the list of analysis points for `st`, specified as:

- Character vector or string — Analysis point identifier that can be any of the following:
  - Signal name, for example `pt = 'torque'`
  - Block path for a block with a single output port, for example `pt = 'Motor/PID'`
  - Block path and port originating the signal, for example `pt = 'Engine Model/1'`
- Cell array of character vectors or string array — Specifies multiple analysis point identifiers. For example:

  ```
  pt = {'torque','Motor/PID','Engine Model/1'}
  ```
- Vector of linearization I/O objects — Create `pt` using `linio`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1,'input');
  pt(2) = linio('scdcascade/Sum',1,'output');
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output.

The interface adds all the points specified by `pt` and ignores their I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings on page 13-490.

`param` — Parameter samples

structure | structure array

Parameter samples for linearizing `mdl`, specified as:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  params(2).Name = 'b';
  params(2).Value = b_grid;
  ```

For more in information, see “Specify Parameter Samples for Batch Linearization” on page 3-62.

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `st.OperatingPoints` with operating point objects only, the software uses single model compilation.

For an example showing how batch linearization with parameter sampling works, see "Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-32. That example uses `slLinearizer`, but the process is the same for `slTuner`.

To compute the offsets required by the LPV System block, specify `param`, and set `st.Options.StoreOffsets` to `true`. You can then return additional linearization information when calling linearization functions such as `getIOTransfer`, and extract the offsets using `getOffsetsForLPV`.

`op` — Operating point for linearizing `mdl`

operating point object | array of operating point objects | vector of positive scalars

Operating point for linearizing `mdl`, specified as:

- Operating point object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of operating point objects, specifying multiple operating points.

  To create an array of operating point objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see "Batch Compute Steady-State Operating Points for Multiple Specifications" on page 1-61.
  - Batch trim your model using parameter variations. For more information, see "Batch Compute Steady-State Operating Points for Parameter Variation" on page 1-65.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `st.Parameters`, then specify `op` as one of the following:

- Single operating point.
- Array of operating point objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see "Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations" on page 3-25.

- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`blocksub` — Substitute linearizations for blocks and model subsystems

structure | structure array

Substitute linearizations for blocks and model subsystems, specified as a structure or an n-by-1 structure array, where n is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

You can batch linearize your model by specifying multiple substitute linearizations for a block. Use this functionality, for example, to study the effects of varying the linearization of a Saturation block on the model dynamics.

Each substitute linearization structure has the following fields:

`Name` — Block path

Block path of the block for which you want to specify the linearization, specified as a character vector.

`Value` — Substitute linearization

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.
- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.
- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:

- Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
- Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for more than one block, the dimensions of the arrays must match.

- Structure with the following fields:

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following • MATLAB expression • Name of a "Custom Linearization Function" on page 13-490 in your current working folder or on the MATLAB path. The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.

Field	Description
`ParameterValues`	Linearization function parameter values, specified as a vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`options` — `slTuner` options
`slTunerOptions` option set

`slTuner` options, specified as an `slTunerOptions` option set.

Example: `options = slTunerOptions('IgnoreDiscreteStates','on')`

Properties

`slTuner` object properties include:

`TunedBlocks`

Blocks to be tuned in `mdl`, specified as a cell array of character vectors.

When you create an `slTuner` interface, the `TunedBlocks` property is automatically populated with the blocks you specify in the `tuned_blocks` input argument. To specify additional tunable blocks in an existing `slTuner` interface, use `addBlock`.

`Ts`

Sampling time for analyzing and tuning `mdl`, specified as nonnegative scalar.

Set this property using dot notation (`st.Ts = Ts`).

Default: 0 (implies continuous-time)

`Parameters`

Parameter samples for linearizing `mdl`, specified as a structure or a structure array.

Set this property using the `param` input argument or dot notation (`st.Parameters = param`). `param` must be one of the following:

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `st.OperatingPoints` with operating point objects only, the software uses single model compilation.

`OperatingPoints`

Operating points for linearizing `mdl`, specified as an operating point object, array of operating point objects, or array of positive scalars.

Set this property using the `op` input argument or dot notation (`st.OperatingPoints = op`). `op` must be one of the following:

- Operating point object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of operating point objects, specifying multiple operating points.

  To create an array of operating point objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-61.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-65.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `st.Parameters`, then specify `op` as one of the following:

- Single operating point.
- Array of operating point objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-25.

- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`BlockSubstitutions`

Substitute linearizations for blocks and model subsystems, specified as a structure or structure array.

Use this property to specify a custom linearization for a block or subsystem. You also can use this syntax for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

Set this property using the `blocksub` input argument or dot notation (`st.BlockSubstitutions = blocksubs`). For information about the required structure, see `blocksub`.

`Options`

Linearization algorithm options, specified as an option set created using `slTunerOptions`.

Set this property using the `opt` input argument or dot notation (`st.Options = opt`).

`Model`

Name of the Simulink model to be linearized, specified as a character vector by the input argument `mdl`.

`TimeUnit`

Unit of the time variable. This property specifies the time units for linearized models returned by `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use any of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`

- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Default: `'seconds'`

Object Functions

addBlock	Add block to list of tuned blocks for slTuner interface
addPoint	Add signal to list of analysis points for slLinearizer or slTuner interface
addOpening	Add signal to list of openings for slLinearizer or slTuner interface
getPoints	Get list of analysis points for slLinearizer or slTuner interface
getOpenings	Get list of openings for slLinearizer or slTuner interface
getBlockParam	Get parameterization of tuned block in slTuner interface
getBlockValue	Get current value of tuned block parameterization in slTuner interface
getTunedValue	Get current value of tuned variable in slTuner interface
getBlockRateConversion	Get rate conversion settings for tuned block in slTuner interface
setBlockParam	Set parameterization of tuned block in slTuner interface
setBlockValue	Set value of tuned block parameterization in slTuner interface
setBlockRateConversion	Set rate conversion settings for tuned block in slTuner interface
systune	Tune control system parameters in Simulink using slTuner interface
looptune	Tune MIMO feedback loops in Simulink using slTuner interface
showTunable	Show value of parameterizations of tunable blocks of slTuner interface
getIOTransfer	Transfer function for specified I/O set using slLinearizer or slTuner interface
getLoopTransfer	Open-loop transfer function at specified point using slLinearizer or slTuner interface

getSensitivity	Sensitivity function at specified point using slLinearizer or slTuner interface
getCompSensitivity	Complementary sensitivity function at specified point using slLinearizer or slTuner interface
writeBlockValue	Update block values in Simulink model
writeLookupTableData	Update portion of tuned lookup table

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-48 and “Mark Signals of Interest for Batch Linearization” on page 3-13.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

Field	Description
`BlockName`	Name of the block for which you are specifying a custom linearization.
`Parameters`	Block parameter values, specified as a structure array with `Name` and `Value` fields. `Parameters` contains the names and values of the parameters you specify in the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Inputs</td><td>Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in Inputs has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>BlockName</td><td>Full block path of the block whose output connects to the corresponding block input.</td></tr>
<tr><td>PortIndex</td><td>Output port of the block specified by BlockName that connects to the corresponding block input.</td></tr>
<tr><td>Values</td><td>Value of the signal specified by BlockName and PortIndex. If this signal is a vector signal, then Values is a vector with the same dimension.</td></tr>
</table></td></tr>
<tr><td>ny</td><td>Number of output channels of the block linearization.</td></tr>
<tr><td>nu</td><td>Number of input channels of the block linearization.</td></tr>
<tr><td>BlockLinearization</td><td>Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using BlockLinearization.</td></tr>
</table>

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

Algorithms

`slTuner` linearizes your Simulink model using the algorithms described in “Exact Linearization Algorithm” on page 2-207.

See Also

`addOpening` | `addPoint` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize` | `looptune` | `systune`

Topics

“Mark Signals of Interest for Control System Analysis and Design” on page 2-48
“How the Software Treats Loop Openings” on page 2-39
“Create and Configure slTuner Interface to Simulink Model” (Control System Toolbox)
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-32
“Tune Control Systems in Simulink” (Control System Toolbox)
“Fault-Tolerant Control of a Passenger Jet” (Control System Toolbox)
“PID Tuning for Setpoint Tracking vs. Disturbance Rejection” (Control System Toolbox)

Introduced in R2014a

slTunerOptions

Set `slTuner` interface options

Syntax

```
options = slTunerOptions
options = slTunerOptions(Name,Value)
```

Description

`options = slTunerOptions` returns the default `slTuner` interface option set.

`options = slTunerOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Option Set for `slTuner` Interface

Create an option set for an `slTuner` interface that sets the rate conversion method to the Tustin method with prewarping at a frequency of 10 rad/s.

```
options = slTunerOptions('RateConversionMethod','prewarp',...
                         'PreWarpFreq',10);
```

Alternatively, use dot notation to set the values of `options`.

```
options = slTunerOptions;
options.RateConversionMethod = 'prewarp';
options.PreWarpFreq = 10;
```

Input Arguments

Name-Value Pair Arguments

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

Example: `'RateConversionMethod','prewarp'` sets the rate conversion method to the Tustin method with prewarping.

`UseFullBlockNameLabels` — Flag indicating whether to truncate names of I/Os and states

`'off'` (default) | `'on'`

Flag indicating whether to truncate names of I/Os and states in the linearized model, specified as the comma-separated pair consisting of `'UseFullBlockNameLabels'` and either:

- `'off'` — Use truncated names for the I/Os and states in the linearized model.
- `'on'` — Use the full block path to name the I/Os and states in the linearized model.

`UseBusSignalLabels` — Flag indicating whether to use bus signal channel numbers or names

`'off'` (default) | `'on'`

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as the comma-separated pair consisting of `'UseBusSignalLabels'` and one of the following:

- `'off'` — Use bus signal channel numbers to label I/Os on bus signals in the linearized model.

- `'on'` — Use bus signal names to label I/Os on bus signals in the linearized model. Bus signal names appear in the results when the I/O points are located at the output of the following blocks:
  - Root-level inport block containing a bus object
  - Bus creator block
  - Subsystem block whose source traces back to the output of a bus creator block
  - Subsystem block whose source traces back to a root-level inport by passing through only virtual or nonvirtual subsystem boundaries

`StoreOffsets` — Flag indicating whether to compute linearization offsets
`false` (default) | `true`

Flag indicating whether to compute linearization offsets for inputs, outputs, states, and state derivatives or updated states, specified as the comma-separated pair consisting of `'StoreOffsets'` and one of the following:

- `false` — Do not compute linearization offsets.
- `true` — Compute linearization offsets.

You can configure an LPV System block using linearization offsets. For an example, see “Approximating Nonlinear Behavior Using an Array of LTI Systems” on page 3-91

`StoreAdvisor` — Flag indicating whether to store diagnostic information
`false` (default) | `true`

Flag indicating whether to store diagnostic information during linearization, specified as the comma-separated pair consisting of `'StoreAdvisor'` and one of the following:

- `false` — Do not store linearization diagnostic information.
- `true` — Store linearization diagnostic information.

Linearization commands store and return diagnostic information in a `LinearizationAdvisor` object. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-40.

`RateConversionMethod` — Rate conversion method
`'zoh'` (default) | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`

Method used for rate conversion when linearizing a multirate system, specified as the comma-separated pair consisting of `'RateConversionMethod'` and one of the following:

- `'zoh'` — Zero-order hold rate conversion method
- `'tustin'` — Tustin (bilinear) method
- `'prewarp'` — Tustin method with frequency prewarp. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.
- `'upsampling_zoh'` — Upsample discrete states when possible, and use `'zoh'` otherwise.
- `'upsampling_tustin'` — Upsample discrete states when possible, and use `'tustin'` otherwise.
- `'upsampling_prewarp'` — Upsample discrete states when possible, and use `'prewarp'` otherwise. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.

For more information on rate conversion and linearization of multirate models, see:

- “Linearization of Multirate Models”.
- “Linearization Using Different Rate Conversion Methods”.
- “Continuous-Discrete Conversion Methods” (Control System Toolbox) .

Note If you use a rate conversion method other than `'zoh'`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

`PreWarpFreq` — Prewarp frequency

`0` (default) | positive scalar

Prewarp frequency in rad/s, specified as the comma-separated pair consisting of `'PreWarpFreq'` and a nonnegative scalar. This option applies only when `RateConversionMethod` is either `'prewarp'` or `'upsampling_prewarp'`.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values

`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for linearization, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.
- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

For more information about model compilation when you linearize with parameter variation, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-10.

Output Arguments

`options` — `slTuner` interface options
`slTunerOptions` option set

`slTuner` interface options, returned as an `slTunerOptions` option set.

See Also

`slTuner`

Introduced in R2014a

update

Update operating point object with structural changes in model

Syntax

```
update(op)
```

Description

`update(op)` updates an operating point object, `op`, to reflect any changes in the associated Simulink model, such as states being added or removed.

Examples

Open the magball model:

```
magball
```

Create an operating point object for the model:

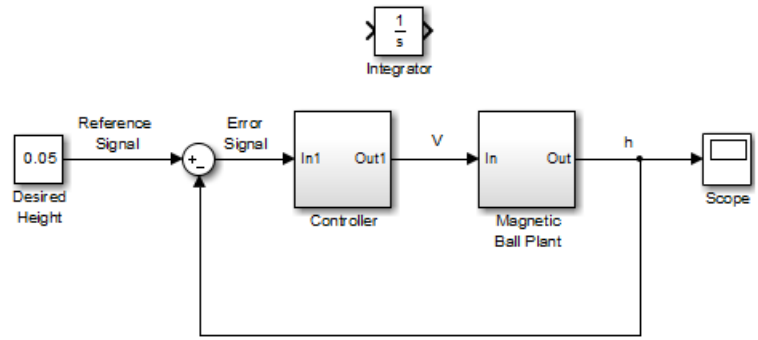
```
op=operpoint('magball')
```

This syntax returns:

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
```

```
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

Add an Integrator block to the model, as shown in the following figure.



Update the operating point to include this new state:

```
update(op)
```

The new operating point appears:

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
(6.) magball/Integrator
      x: 0
```

```
Inputs: None
----------
```

Alternatives

As an alternative to the `update` function, update operating point objects using the **Sync with Model** button in the Linear Analysis Tool.

See Also

`operpoint` | `operspec`

Introduced before R2006a

writeLookupTableData

Update portion of tuned lookup table

When tuning lookup table blocks with `systune`, use this function to update only a portion of the table data in the Simulink model. This function is useful when retuning a single point or a portion of the lookup table. To update the entire lookup table, use `writeBlockValue`.

Syntax

```
writeLookupTableData(st,blockid,breakpoints)
writeLookupTableData(st,blockid,ix1,…,ixN)
```

Description

`writeLookupTableData(st,blockid,breakpoints)` writes tuned gain values from an `slTuner` interface to a portion of a lookup table in the associated Simulink model. Each row of `breakpoints` identifies an entry in the lookup table to update. `blockid` must identify a single block in the `TunedBlocks` property of the `slTuner` interface.

`writeLookupTableData(st,blockid,ix1,…,ixN)` updates a rectangular portion of the table data. The index vectors `ix1,…,ixN` select specific breakpoints along each table dimension.

Examples

Update Specific Entries in Lookup Table

Suppose you have an `slTuner` interface `st` to a Simulink model that contains a 2-D Lookup Table block `Kp Lookup`. The block is listed in `slTuner.TunedBlocks`. Suppose further that you have retuned for design points corresponding to the (3,5) and (4,6) breakpoints in the lookup table. Update the lookup table with the new values.

```
breakpoints = [3 5;4 6];
writeLookupTableData(st,'Kp Lookup',breakpoints)
```

Update Rectangular Portion of Lookup Table

Suppose you have retuned design points between the third and fifth values of the first scheduling variable, and the seventh and tenth values of the second scheduling variables. Update the lookup table with the new values.

```
ix1 = 3:5;
ix2 = 7:10;
writeLookupTableData(st,'Kp Lookup',ix1,ix2)
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blockid` — Lookup table

character vector | string

Lookup table to update with tuned values, specified as a character vector or string. The block identified by `blockid` must be a lookup-table block in the `TunedBlocks` property of the `slTuner` interface `st`. You can specify a full block path, or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `'scdcascade/Kp Lookup'`

Example: `"Kp Lookup"`

`breakpoints` — Lookup-table entries

integer array

Lookup-table entries to update, specified as an integer array. Each row of `breakpoints` specifies a table entry by its `(i1,i2,…,iN)` subscripts. For instance:

- To update the data associated with the first and third breakpoints in a 1-D Lookup Table block, use `breakpoints = [1;3]`.
- To update the data associated with the (3,5) and (4,6) entries in a 2-D Lookup Table block, use `breakpoints = [3 5;4 6]`.

`ix1,…,ixN` — Portion of lookup table

vectors

Portion of lookup table to update, specified as index vectors that select specific breakpoints along each table dimension. For instance, to update a 2-D Lookup Table block, specify two index vectors that identify the rows and columns to update. If you want to update the portion of the table blocked out by entries 3 through 5 in the first dimension and 7 through 10 in the second dimension, use `ix1 = 3:5` and `ix2 = 7:10`.

Tips

- If you use `writeBlockValue` to update other retuned blocks in your model, exclude the lookup table `blockid` from the list of blocks to update with that function.

See Also

`writeBlockValue`

Topics

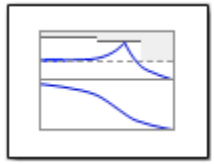
“Validate Gain-Scheduled Control Systems” on page 10-46

Introduced in R2017b

Blocks — Alphabetical List

Bode Plot

Bode plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Bode Characteristics block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a nonlinear Simulink model and plot the linear system on a Bode plot.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the magnitude and phase of the linear system.

The Simulink model can be continuous- or discrete-time or multirate, and can have time delays. The linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO). For MIMO systems, the plots for all input/output combinations are displayed.

You can specify piecewise-linear frequency-dependent upper and lower magnitude bounds and view them on the Bode plot. You can also check that the bounds are satisfied during simulation:

- If all bounds are satisfied, the block does nothing.

- If a bound is not satisfied, the block asserts, and a warning message appears at the MATLAB prompt. You can also specify that the block:
  - Evaluate a MATLAB expression.
  - Stop the simulation and bring that block into focus.

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

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

For MIMO systems, the bounds apply to the Bode responses of linear systems computed for all input/output combinations.

You can add multiple Bode Plot blocks to compute and plot the magnitude and phase of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Bode Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/outputs" on page 14-5. • "Click a signal in the model to select it" on page 14-8.

Task		Parameters
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-10. • "Snapshot times" on page 14-11. • "Trigger type" on page 14-12.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-12. • "Use exact delays" on page 14-14. • "Linear system sample time" on page 14-14. • "Sample time rate conversion method" on page 14-15. • "Prewarp frequency (rad/s)" on page 14-17.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-18. • "Use bus signal names" on page 14-19.
Plot the linear system.		**Show Plot** on page 14-35
(Optional) Specify bounds on magnitude of the linear system for assertion.		In **Bounds** tab: • "Include upper magnitude bound in assertion" on page 14-20. • "Include lower magnitude bound in assertion" on page 14-24.

Task	Parameters
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-30. • "Simulation callback when assertion fails (optional)" on page 14-32. • "Stop simulation when assertion fails" on page 14-32. • "Output assertion signal" on page 14-33.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-27 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-34.

Linearization inputs/outputs

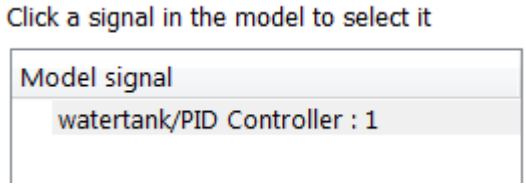
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click  .

   The dialog box expands to display a **Click a signal in the model to select it** on page 14-8 area and a new  button.
2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

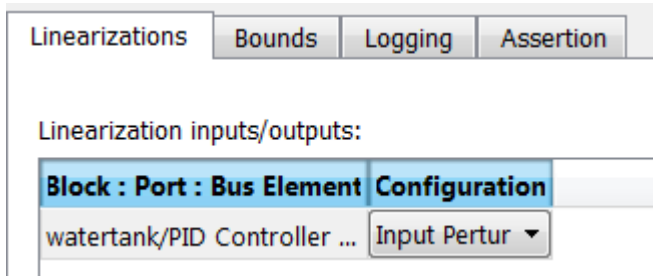
To modify the filtering options, click . To hide the filtering options, click .

Filtering Options

- "Enable regular expression" on page 14-8
- "Show filtered results as a flat list" on page 14-9

---

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

The table displays the following information about the selected signal:

Block : Port : Bus Element Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click  .

When this option appears, you also see the following changes:

- A new  button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-5.
-  changes to  .

  Use  to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

 On

Allow use of MATLAB regular expressions for filtering signal names.

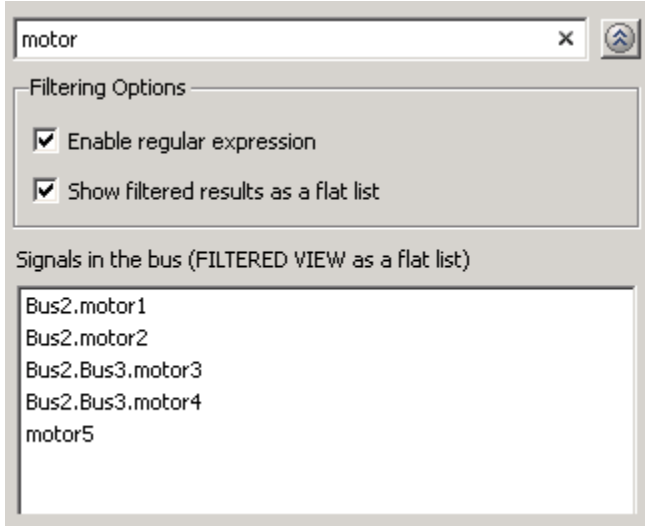
Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box ( ) enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-11.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-12.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-10 enables this parameter.

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-10 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising' | 'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

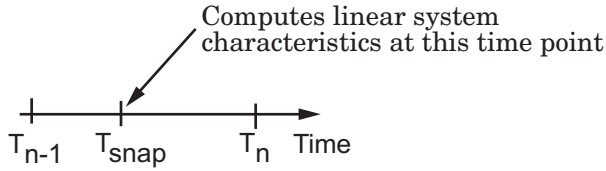
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

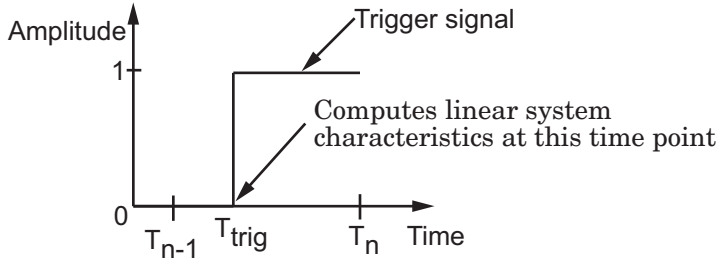
- The exact snapshot times, specified in **Snapshot times** on page 14-11.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-12.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

☐ Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

 On

Return a linear model with exact delay representations.

 Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-15.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-14 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-17.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-15 enables this parameter.

Parameter: `PreWarpFreq`

Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

 On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

 Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

Parameter: `UseFullBlockNameLabels`
Type: character vector

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

 On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include upper magnitude bound in assertion

Check that the Bode response satisfies upper magnitude bounds, specified in **Frequencies (rad/sec)** on page 14-21 and **Magnitude (dB)** on page 14-22, during simulation. The software displays a warning if the magnitude violates the upper bounds.

This parameter is used for assertion only if **Enable assertion** on page 14-30 in the **Assertion** tab is selected.

You can specify multiple upper magnitude bounds on the linear system. The bounds also appear on the Bode magnitude plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default:

- Off for Bode Plot block.
- On for Check Bode Characteristics block.

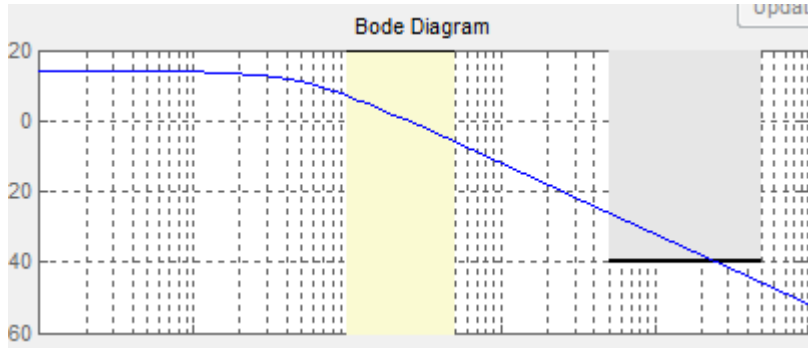
 On

Check that the magnitude satisfies the specified upper bounds, during simulation.

 Off

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

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



- If you specify both upper and lower magnitude bounds on page 14-24 but want to include only the lower bounds for assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Parameter: `EnableUpperBound`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Frequencies (rad/sec)

Frequencies for one or more upper magnitude bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 14-22.

Default:
`[]` for Bode Plot block
`[10 100]` for Check Bode Characteristics block

Must be specified as start and end frequencies:

- Positive finite numbers for a single bound with one edge

- Matrix of positive finite numbers for a single bound with multiple edges

  For example, type [0.1 1;1 10] for two edges at frequencies [0.1 1] and [1 10].
- Cell array of matrices with positive finite numbers for multiple bounds

- To assert that magnitudes that correspond to the frequencies are satisfied, select both **Include upper magnitude bound in assertion** on page 14-20 and **Enable assertion** on page 14-30.
- You can add or modify frequencies from the plot window:
  - To add new frequencies, right-click the plot, and select **Bounds** > **New Bound**. Select `Upper gain limit` in **Design requirement type**, and specify the frequencies in the **Frequency** column. Specify the corresponding magnitudes in the **Magnitude** column.
  - To modify the frequencies, drag the bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit Bound**. Specify the new frequencies in the **Frequency** column.

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

Parameter: `UpperBoundFrequencies`
Type: character vector
Value: `[]` | `[10 100]` | positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Bode Plot block, `'[10 100]'` for Check Bode Characteristics block

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Magnitudes (dB)

Magnitude values for one or more upper magnitude bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 14-21.

Default:
`[]` for Bode Plot block
`[-20 -20]` for Check Bode Characteristics block

Must be specified as start and end magnitudes:

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

  For example, type [–10 –10; –20 –20] for two edges at magnitudes [–10 –10] and [–20 –20].
- Cell array of matrices with finite numbers for multiple bounds

- To assert that magnitude bounds are satisfied, select both **Include upper magnitude bound in assertion** on page 14-20 and **Enable assertion** on page 14-30.
- You can add or modify magnitudes from the plot window:
  - To add a new magnitude, right-click the plot, and select **Bounds > New Bound**. Select `Upper gain limit` in **Design requirement type**, and specify the magnitude in the **Magnitude** column. Specify the corresponding frequencies in the **Frequency** column.
  - To modify the magnitudes, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bound**. Specify the new magnitudes in the **Magnitude** column.

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

Parameter: `UpperBoundMagnitudes`
Type: character vector
Value: `[]` | `[-20 -20]` | finite number | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Bode Plot block, `'[-20 -20]'` for Check Bode Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include lower magnitude bound in assertion

Check that the Bode response satisfies lower magnitude bounds, specified in **Frequencies (rad/sec)** on page 14-25 and **Magnitude (dB)** on page 14-26, during simulation. The software displays a warning if the magnitude violates the lower bounds.

This parameter is used for assertion only if **Enable assertion** on page 14-30 in the **Assertion** tab is selected.

You can specify multiple lower magnitude bounds on the linear system computed during simulation. The bounds also appear on the Bode magnitude plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default:

- Off for Bode Plot block.
- On for Check Bode Characteristics block

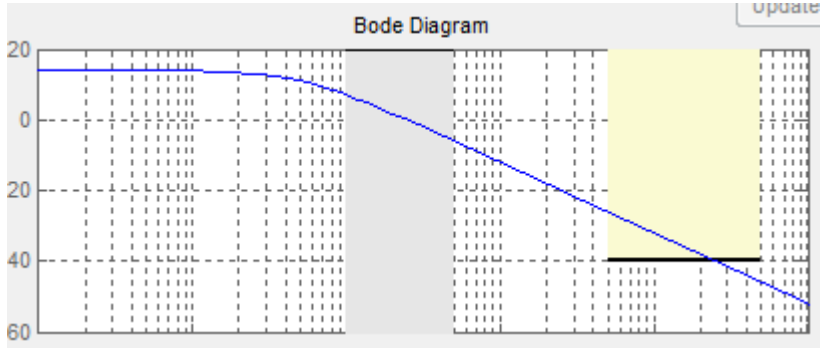
☑ On

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

☐ Off

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

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



- If you specify both upper and lower magnitude bounds on the Bode magnitude but want to include only the upper bound for assertion, clear this parameter.
- To only view the bound on the plot, clear **Enable assertion**.

Parameter: `EnableLowerBound`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Frequencies (rad/sec)

Frequencies for one or more lower magnitude bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 14-26.

Default:
`[]` for Bode Plot block
`[0.1 1]` for Check Bode Characteristics block

Must be specified as start and end frequencies:

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

  For example, type [0.1 1;1 10] to specify two edges with frequencies [0.1 1] and [1 10].
- Cell array of matrices with positive finite numbers for multiple bounds

- To assert that magnitude bounds that correspond to the frequencies are satisfied, select both **Include lower magnitude bound in assertion** on page 14-24 and **Enable assertion** on page 14-30.
- You can add or modify frequencies from the plot window:
  - To add a new frequencies, right-click the plot, and select **Bounds > New Bound**. Select `Lower gain limit` in **Design requirement type**, and specify the

frequencies in the **Frequency** column. Specify the corresponding magnitudes in the **Magnitude** column.

- To modify the frequencies, drag the bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit Bound**. Specify the new frequencies in the **Frequency** column.

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

Parameter: `LowerBoundFrequencies`
Type: character vector
Value: `[]` | `[0.1 1]` | positive finite number | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Bode Plot block, `'[0.1 1]'` for Check Bode Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Magnitudes (dB)

Magnitude values for one or more lower magnitude bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 14-25.

Default:
`[]` for Bode Plot block
`[20 20]` for Check Bode Characteristics block

Must be specified as start and end magnitudes:

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

  For example, type [20 20; 40 40] for two edges with magnitudes [20 20] and [40 40].
- Cell array of matrices with finite numbers for multiple bounds

- To assert that magnitude bounds are satisfied, select both **Include lower magnitude bound in assertion** on page 14-24 and **Enable assertion** on page 14-30.
- If **Include lower magnitude bound in assertion** is not selected, the bound segment is disabled on the plot.
- To only view the bound on the plot, clear **Enable assertion**.
- You can add or modify magnitudes from the plot window:
  - To add a new magnitude, right-click the plot, and select **Bounds > New Bound**. Select `Lower gain limit` in **Design requirement type** and specify the magnitude in the **Magnitude** column. Specify the corresponding frequencies in the **Frequency** column.
  - To modify the magnitudes, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bound**. Specify the new magnitude values in the **Magnitude** column.

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

Parameter: `LowerBoundMagnitudes`
Type: character vector
Value: `[]` | `[20 20]` | finite number | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Bode Plot block, `'[20 20]'` for Check Bode Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.

- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-29.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-27 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

 On

Save the operating points.

 Off

Do not save the operating points.

Save data to workspace on page 14-27 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

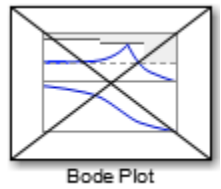
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-32.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-32.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

☑ On

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

☐ Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-30 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

On

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

Off

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

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

Enable assertion on page 14-30 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

On

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

Off

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking  or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking  or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.

- Simulating the model by clicking  or selecting **Simulation** > **Start**. This action also linearizes the portion of the model between the specified linearization input and output.
- Adding legend on the linear system characteristic plot by clicking  .

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools** > **Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

- "Design Optimization to Meet Step Response Requirements (GUI)" (Simulink Design Optimization)
- "Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)" (Simulink Design Optimization)

See Also

Check Bode Characteristics

Tutorials

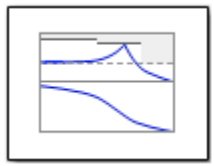
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-110
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-117

- “Plotting Linear System Characteristics of a Chemical Reactor”

Introduced in R2010b

Check Bode Characteristics

Check that Bode magnitude bounds are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Bode Plot block except for different default parameter settings in the **Bounds** tab.

Check that upper and lower magnitude bounds on the Bode response of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multi-rate and can have time delays. The computed linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO).

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, computes the Bode magnitude and phase, and checks that the magnitude satisfies the specified bounds.

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

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

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

For MIMO systems, the bounds apply to the Bode responses computed for all input/output combinations.

You can add multiple Check Bode Characteristics blocks in your model to check upper and lower Bode magnitude bounds on various portions of the model.

You can also plot the magnitude and phase on a Bode plot and graphically verify that the magnitude satisfies the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Check Bode Characteristics block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-3 in the Bode Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on the linear system for assertion.		In **Bounds** tab: • **Include upper magnitude bound in assertion** • **Include lower magnitude bound in assertion**
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**

Task	Parameters
Save linear system to MATLAB workspace.	**Save data to workspace** in **Logging** tab.
View bounds violations graphically in a plot window.	**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.	**Show plot on block open**

See Also

Bode Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

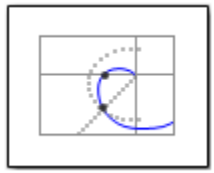
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Check Gain and Phase Margins

Check that gain and phase margin bounds are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Gain and Phase Margin Plot block except for different default parameter settings in the **Bounds** tab.

Check that bounds on gain and phase margins of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, computes the gain and phase margins, and checks that the gain and phase margins satisfy the specified bounds.

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

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

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

You can add multiple Check Gain and Phase Margins blocks in your model to check gain and phase margin bounds on various portions of the model.

You can also plot the gain and phase margins on a Bode, Nichols or Nyquist plot or view the margins in a table and verify that the gain and phase margins satisfy the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Gain and Phase Margin Plot block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-63 in the Gain and Phase Margin Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**

Task		Parameters
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on gain and phase margins of the linear system for assertion.		**Include gain and phase margins in assertion** in **Bounds** tab.
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**
Save linear system to MATLAB workspace.		**Save data to workspace** in **Logging** tab.
View bounds violations graphically in a plot window.		**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.		**Show plot on block open**

See Also

Gain and Phase Margin Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

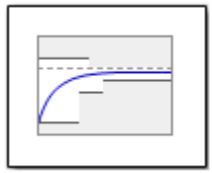
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Check Linear Step Response Characteristics

Check that step response bounds on linear system are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Linear Step Response Plot block except for different default parameter settings in the **Bounds** tab.

Check that bounds on step response characteristics of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, computes the step response and checks that the step response satisfies the specified bounds:

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

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

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

You can add multiple Check Linear Step Response Characteristics blocks in your model to check step response bounds on various portions of the model.

You can also plot the step response and graphically verify that the step response satisfies the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Linear Step Response Plot block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-96 in the Linear Step Response Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**

Task		Parameters
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on the linear system for assertion.		**Include step response bounds in assertion** in **Bounds** tab.
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**
Save linear system to MATLAB workspace.		**Save data to workspace** in **Logging** tab.
View bounds violations graphically in a plot window.		**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.		**Show plot on block open**

See Also

Linear Step Response Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

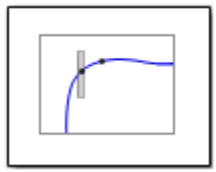
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Check Nichols Characteristics

Check that gain and phase bounds on Nichols response are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Nichols Plot block except for different default parameter settings in the **Bounds** tab.

Check that open- and closed-loop gain and phase bounds on Nichols response of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, computes the magnitude and phase, and checks that the gain and phase satisfy the specified bounds:

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

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

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

You can add multiple Check Nichols Characteristics blocks in your model to check gain and phase bounds on various portions of the model.

You can also plot the linear system on a Nichols plot and graphically verify that the Nichols response satisfies the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Nichols Plot block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-132 in the Nichols Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**

Task		Parameters
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on gains and phases of the linear system for assertion.		In **Bounds** tab: • **Include gain and phase margins in assertion** • **Include closed-loop peak gain in assertion** • **Include open-loop gain-phase bound in assertion**
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**
Save linear system to MATLAB workspace.		**Save data to workspace** in **Logging** tab.

Task	Parameters
View bounds violations graphically in a plot window.	**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.	**Show plot on block open**

See Also

Nichols Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

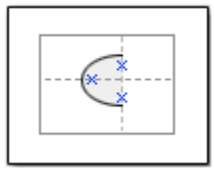
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Check Pole-Zero Characteristics

Check that bounds on pole locations are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Pole-Zero Plot block except for different default parameter settings in the **Bounds** tab.

Check that approximate second-order bounds on the pole locations of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, computes the poles and zeros, and checks that the poles satisfy the specified bounds:

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

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

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

You can add multiple Check Pole-Zero Characteristics blocks in your model to check approximate second-order bounds on various portions of the model.

You can also plot the poles and zeros on a pole-zero map and graphically verify that the poles satisfy the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Pole-Zero Plot block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-190 in the Pole-Zero Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**

Task		Parameters
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on the linear system for assertion.		In **Bounds** tab: • **Include settling time bound in assertion** • **Include percent overshoot bound in assertion** • **Include damping ratio bound in assertion** • **Include natural frequency bound in assertion**
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**

Task	Parameters
Save linear system to MATLAB workspace.	**Save data to workspace** in **Logging** tab.
View bounds violations graphically in a plot window.	**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.	**Show plot on block open**

See Also

Pole-Zero Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

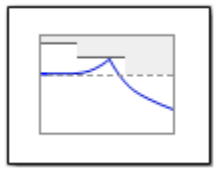
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Check Singular Value Characteristics

Check that singular value bounds are satisfied during simulation



Library

Simulink Control Design

Description

This block is same as the Singular Value Plot block except for default parameter settings in the **Bounds** tab:

Check that upper and lower bounds on singular values of a linear system, computed from a nonlinear Simulink model, are satisfied during simulation.

The Simulink model can be continuous-time, discrete-time or multi-rate and can have time delays. The computed linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO).

During simulation, the software linearizes the portion of the model between specified linearization input and output, computes the singular values, and checks that the values satisfy the specified bounds:

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

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

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

For MIMO systems, the bounds apply to the singular values computed for all input/output combinations.

You can add multiple Check Singular Value Characteristics blocks in your model to check upper and lower singular value bounds on various portions of the model.

You can also plot the singular values on a singular value plot and graphically verify that the values satisfy the bounds.

This block and the other Model Verification blocks test that the linearized behavior of a nonlinear Simulink model is within specified bounds during simulation.

- When a model does not violate any bound, you can disable the block by clearing the assertion option. If you modify the model, you can re-enable assertion to ensure that your changes do not cause the model to violate a bound.
- When a model violates any bound, you can use Simulink Design Optimization software to optimize the linear system to meet the specified requirements in this block.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Singular Value Plot block parameters, accessible via the block parameter dialog box. For more information, see “Parameters” on page 14-230 in the Singular Value Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on the linear system for assertion.		In **Bounds** tab: • **Include upper singular value bound in assertion** • **Include lower singular value bound in assertion**
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • **Enable assertion** • **Simulation callback when assertion fails (optional)** • **Stop simulation when assertion fails** • **Output assertion signal**

Task	Parameters
Save linear system to MATLAB workspace.	**Save data to workspace** in **Logging** tab.
View bounds violations graphically in a plot window.	**Show Plot**
Display plot window instead of block parameters dialog box on double-clicking the block.	**Show plot on block open**

See Also

Singular Value Plot

Tutorials

- "Verify Model at Default Simulation Snapshot Time" on page 12-6
- "Verify Model at Multiple Simulation Snapshots" on page 12-15
- "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25
- "Verifying Frequency-Domain Characteristics of an Aircraft"

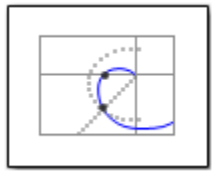
How To

"Monitor Linear System Characteristics in Simulink Models" on page 12-2

Introduced in R2010b

Gain and Phase Margin Plot

Gain and phase margins of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Gain and Phase Margins block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a nonlinear Simulink model and view the gain and phase margins on a Bode, Nichols or Nyquist plot. Alternatively, you can view the margins in a table. By default, the margins are computed using negative feedback for the closed-loop system.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the linear system on the specified plot type.

The Simulink model can be continuous- or discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

You can specify only one gain and phase margin bound each and view them on the selected plot or table. The block does not support multiple gain and phase margin bounds. You can also check that the bounds are satisfied during simulation:

- If all bounds are satisfied, the block does nothing.

- If a bound is not satisfied, the block asserts, and a warning message appears at the MATLAB prompt. You can also specify that the block:
  - Evaluate a MATLAB expression.
  - Stop the simulation and bring that block into focus.

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

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

You can add multiple Gain and Phase Margin Plot blocks to compute and plot the gain and phase margins of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Gain and Phase Margin Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/ outputs" on page 14-65. • "Click a signal in the model to select it" on page 14-68.

Task		Parameters
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-70. • "Snapshot times" on page 14-71. • "Trigger type" on page 14-72.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-72. • "Use exact delays" on page 14-74. • "Linear system sample time" on page 14-75. • "Sample time rate conversion method" on page 14-76. • "Prewarp frequency (rad/s)" on page 14-77.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-78. • "Use bus signal names" on page 14-79.
Specify plot type for viewing gain and phase margins.		"Plot type" on page 14-91.
Plot the linear system.		**Show Plot** on page 14-93
Specify the feedback sign for closed-loop gain and phase margins.		"Feedback sign" on page 14-83 in **Bounds** tab.

Task	Parameters
(Optional) Specify bounds on gain and phase margins of the linear system for assertion.	"Include gain and phase margins in assertion" on page 14-80 in **Bounds** tab.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-87. • "Simulation callback when assertion fails (optional)" on page 14-89. • "Stop simulation when assertion fails" on page 14-89. • "Output assertion signal" on page 14-90.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-84 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-92.

Linearization inputs/outputs

Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

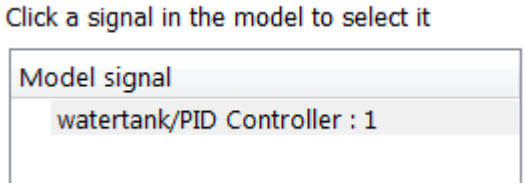
If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1 Click  .

The dialog box expands to display a **Click a signal in the model to select it** on page 14-68 area and a new « button.

2 Select one or more signals in the Simulink Editor.

The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

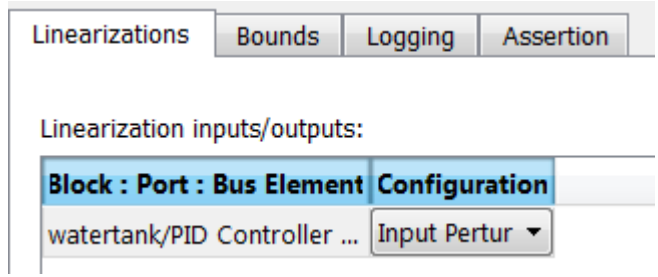
To modify the filtering options, click ⌄⌄. To hide the filtering options, click ⌃⌃.

Filtering Options

- "Enable regular expression" on page 14-68
- "Show filtered results as a flat list" on page 14-69

---

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click  .

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click  .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration — Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click [+].

When this option appears, you also see the following changes:

- A new [«] button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-65.
- [+] changes to [−].

  Use [−] to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a

lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

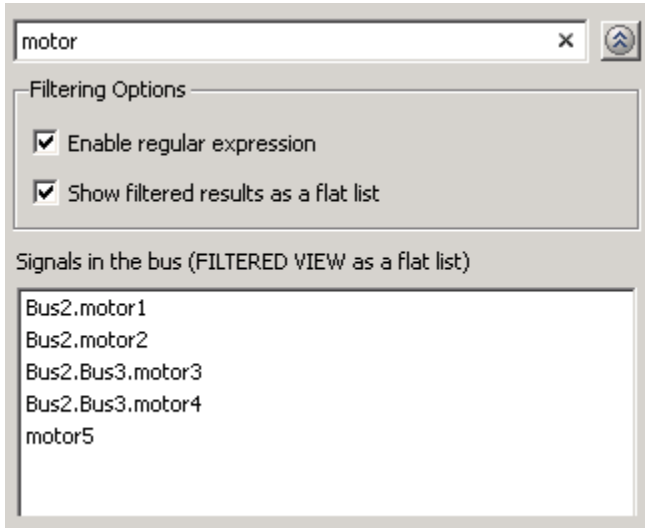
☐ Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

 On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

 Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-71.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-72.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-70 enables this parameter.

Parameter: `SnapshotTimes`

Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-70 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

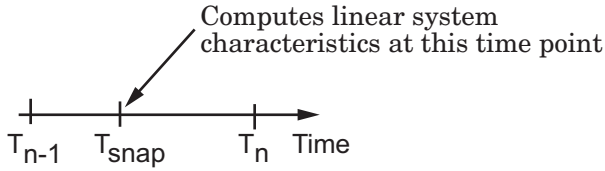
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

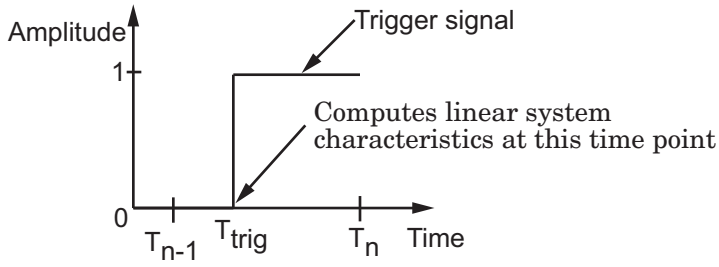
- The exact snapshot times, specified in **Snapshot times** on page 14-71.

As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-72.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

On

Return a linear model with exact delay representations.

Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-76.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-75 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-77.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-76 enables this parameter.

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

☑ On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

☐ Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

☑ On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include gain and phase margins in assertion

Check that the gain and phase margins are greater than the values specified in **Gain margin (dB) >** on page 14-81 and **Phase margin (deg) >** on page 14-82, during simulation. The software displays a warning if the gain or phase margin is less than or equals the specified value.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the margins.

This parameter is used for assertion only if **Enable assertion** on page 14-87 in the **Assertion** tab is selected.

You can view the gain and phase margin bound on one of the following plot types on page 14-91:

- Bode
- Nichols

- Nyquist
- Table

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

Default:

- Off for Gain and Phase Margin Plot block.
- On for Check Gain and Phase Margins block.

 On

Check that the gain and phase margins satisfy the specified values, during simulation.

 Off

Do not check that the gain and phase margins satisfy the specified values, during simulation.

- Clearing this parameter disables the gain and phase margin bounds and the software stops checking that the gain and phase margins satisfy the bounds during simulation. The gain and phase margin bounds are also disabled on the plot.
- To only view the gain and phase margin on the plot, clear **Enable assertion**.

Parameter: `EnableMargins`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Gain and Phase Margin Plot block, `'on'` for Check Gain and Phase Margins block

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Gain margin (dB) >

Gain margin, specified in decibels.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the gain margin.

You can specify only one gain margin bound on the linear system in this block.

Default:
`[]` for Gain and Phase Margin Plot block.
`20` for Check Gain and Phase Margins block.

Positive finite number.

- To assert that the gain margin is satisfied, select both **Include gain and phase margins in assertion** on page 14-80 and **Enable assertion** on page 14-87.
- To modify the gain margin from the plot window, right-click the plot, and select **Bounds** > **Edit Bound**. Specify the new gain margin in **Gain margin >**. You must click **Update Block** before simulating the model.

Parameter: `GainMargin`
Type: character vector
Value: `[]` | `20` | positive finite number. Must be specified inside single quotes (`''`).
Default: `'[]'` for Gain and Phase Margin Plot block, `'20'` for Check Gain and Phase Margins block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Phase margin (deg) >

Phase margin, specified in degrees.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the phase margin.

You can specify only one phase margin bound on the linear system in this block.

Default:

`[]` for Gain and Phase Margin Plot block.
`30` for Check Gain and Phase Margins block.

Positive finite number.

- To assert that the phase margin is satisfied, select both **Include gain and phase margins in assertion** on page 14-80 and **Enable assertion** on page 14-87.
- To modify the phase margin from the plot window, right-click the plot, and select **Bounds** > **Edit Bound**. Specify the new phase margin in **Phase margin >**. You must click **Update Block** before simulating the model.

Parameter: `PhaseMargin`
Type: character vector
Value: `[]` | `30` | positive finite number. Must be specified inside single quotes (`''`).
Default: `'[]'` for Gain and Phase Margin Plot block, `'30'` for Check Gain and Phase Margins block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Feedback sign

Feedback sign to determine the gain and phase margins of the linear system, computed during simulation.

To determine the feedback sign, check if the path defined by the linearization inputs and outputs include the feedback Sum block:

- If the path includes the Sum block, specify positive feedback.
- If the path does not include the Sum block, specify the same feedback sign as the Sum block.

For example, in the aircraft model, the Check Gain and Phase Margins block includes the negative sign in the summation block. Therefore, the **Feedback sign** is positive.

Default: `negative feedback`

`negative feedback`

Use when the path defined by the linearization inputs/outputs *does not include* the Sum block and the Sum block feedback sign is `-`.

`positive feedback`

Use when:

- The path defined by the linearization inputs/outputs *includes* the Sum block.
- The path defined by the linearization inputs/outputs *does not include* the Sum block and the Sum block feedback sign is `+`.

Parameter: `FeedbackSign`
Type: character vector
Value: `'-1' | '+1'`
Default: `'-1'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.

- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation** > **Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-85.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-84 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

 On

Save the operating points.

 Off

Do not save the operating points.

Save data to workspace on page 14-84 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

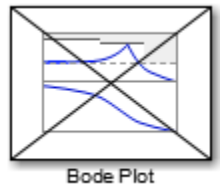
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-89.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-89.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

☑ On

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

☐ Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-87 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

 On

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

Off

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

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

Enable assertion on page 14-87 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Plot type

Plot to view gain and phase margins of the linear system computed during simulation.

Default: `Bode`

`Bode`

Bode plot.

`Nichols`

Nichols plot

`Nyquist`

Nyquist plot

`Tabular`

Table.

Right-click the Bode , Nichols or Nyquist plot and select **Characteristics > Minimum Stability Margins** to view gain and phase margins. The table displays the computed margins automatically.

Parameter: `PlotType`
Type: character vector
Value: `'bode' | 'nichols' | 'nyquist' | 'table'`
Default: `'bode'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

 On

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

 Off

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

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

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools > Response**

Optimization in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

- "Design Optimization to Meet Step Response Requirements (GUI)" (Simulink Design Optimization)
- "Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)" (Simulink Design Optimization)

See Also

Check Gain and Phase Margins

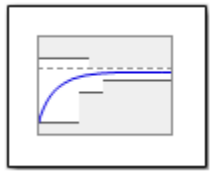
Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-110
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-117
- "Plotting Linear System Characteristics of a Chemical Reactor"

Introduced in R2010b

Linear Step Response Plot

Step response of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Linear Step Response Characteristics block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a nonlinear Simulink model and plot the linear step response.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the step response of the linear system.

The Simulink model can be continuous- or discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

You can specify step response bounds and view them on the plot. You can also check that the bounds are satisfied during simulation:

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

- Evaluate a MATLAB expression.
- Stop the simulation and bring that block into focus.

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

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

You can add multiple Linear Step Response Plot blocks to compute and plot the linear step response of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Linear Step Response Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/ outputs" on page 14-98. • "Click a signal in the model to select it" on page 14-101.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-103. • "Snapshot times" on page 14-104. • "Trigger type" on page 14-105.

Task		Parameters
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-105. • "Use exact delays" on page 14-107. • "Linear system sample time" on page 14-107. • "Sample time rate conversion method" on page 14-108. • "Prewarp frequency (rad/s)" on page 14-110.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-111. • "Use bus signal names" on page 14-112.
Plot the linear system.		**Show Plot** on page 14-128
(Optional) Specify bounds on step response of the linear system for assertion.		Include step response bound in assertion on page 14-113 in **Bounds** tab.

Task	Parameters
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-124. • "Simulation callback when assertion fails (optional)" on page 14-125. • "Stop simulation when assertion fails" on page 14-126. • "Output assertion signal" on page 14-126.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-121 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-127.

Linearization inputs/outputs

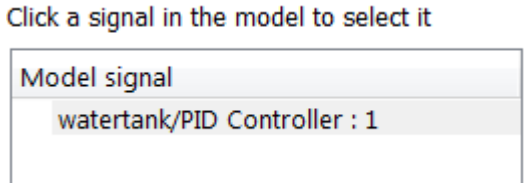
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click  .

   The dialog box expands to display a **Click a signal in the model to select it** on page 14-101 area and a new  button.
2. Select one or more signals in the Simulink Editor.

The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

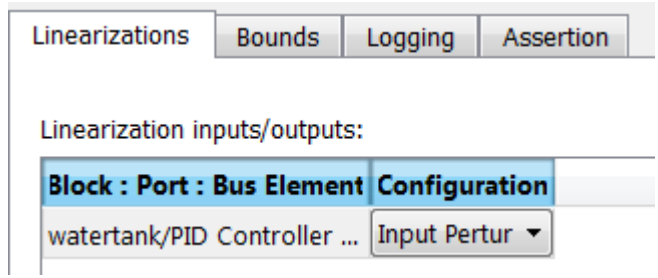
To modify the filtering options, click . To hide the filtering options, click .

Filtering Options

- "Enable regular expression" on page 14-101
- "Show filtered results as a flat list" on page 14-102

---

4 Click to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click .

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

The table displays the following information about the selected signal:

Block : Port : Bus Element	Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.
Configuration	Type of linearization point: • `Open-loop Input` — Specifies a linearization input point after a loop opening. • `Open-loop Output` — Specifies a linearization output point before a loop opening. • `Loop Transfer` — Specifies an output point before a loop opening followed by an input. • `Input Perturbation` — Specifies an additive input to a signal. • `Output Measurement` — Takes measurement at a signal. • `Loop Break` — Specifies a loop opening. • `Sensitivity` — Specifies an additive input followed by an output measurement. • `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click  .

When this option appears, you also see the following changes:

- A new  button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-98.
-  changes to  .

  Use  to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

 On

Allow use of MATLAB regular expressions for filtering signal names.

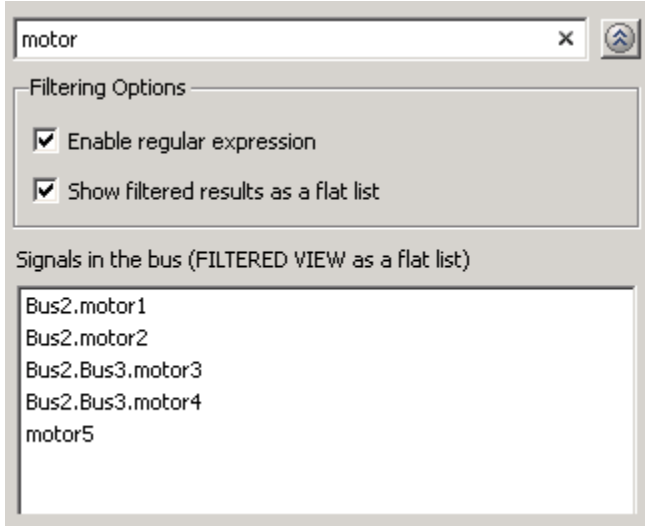
Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box ( ) enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-104.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-105.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-103 enables this parameter.

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-103 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising' | 'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

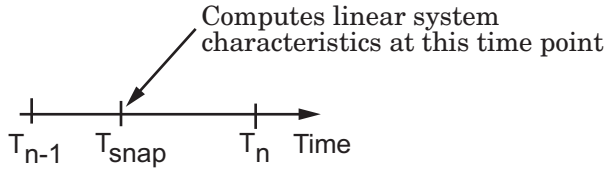
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

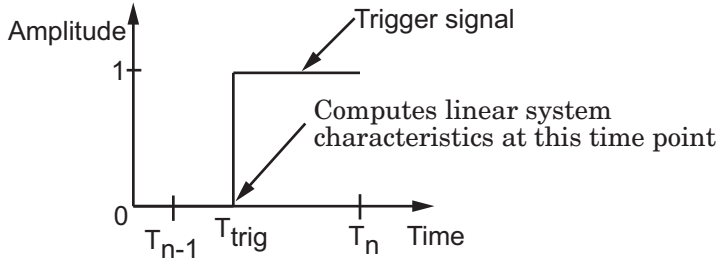
- The exact snapshot times, specified in **Snapshot times** on page 14-104.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-105.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

☐ Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

 On

Return a linear model with exact delay representations.

 Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-108.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-107 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-110.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-108 enables this parameter.

Parameter: `PreWarpFreq`

Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

 On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

 Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

Parameter: `UseFullBlockNameLabels`
Type: character vector

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

 On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include step response bound in assertion

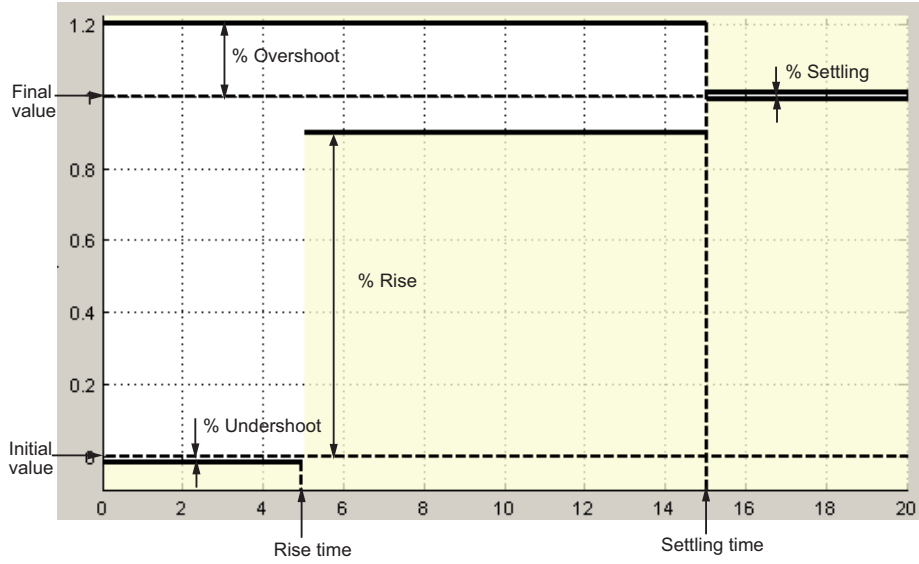
Check that the linear step response satisfies *all* the characteristics specified in:

- **Final value** on page 14-115
- **Rise time** on page 14-116 and **% Rise** on page 14-116
- **Settling time** on page 14-117 and **% Settling** on page 14-118
- **% Overshoot** on page 14-119
- **% Undershoot** on page 14-120

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

This parameter is used for assertion only if **Enable assertion** on page 14-124 in the **Assertion** tab is selected.

The bounds also appear on the step response plot, as shown in the next figure.



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

Default:

- Off for Linear Step Response Plot block.
- On for Check Linear Step Response Characteristics block.

On

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

Off

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

- Clearing this parameter disables the step response bounds and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.
- To only view the bounds on the plot, clear **Enable assertion**.

Parameter: `EnableStepResponseBound`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Linear Step Response Plot block, `'on'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Final value

Final value of the output signal level in response to a step input.

Default:

- `[]` for Linear Step Response Plot block
- `1` for Check Linear Step Response Characteristics block

Finite real scalar.

- To assert that final value is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the final value from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit**. Specify the new value in **Final value**. You must click **Update Block** before simulating the model.

Parameter: `FinalValue`
Type: character vector
Value: `[] | 1` | finite real scalar. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Rise time

Time taken, in seconds, for the step response to reach a percentage of the final value specified in **% Rise** on page 14-116.

Default:

- `[]` for Linear Step Response Plot block
- `5` for Check Linear Step Response Characteristics block

Finite positive real scalar, less than the settling time on page 14-117.

- To assert that the rise time is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the rise time from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds > Edit**. Specify the new value in **Rise time**. You must click **Update Block** before simulating the model.

Parameter: `RiseTime`
Type: character vector
Value: `[] | 5 |` finite positive real scalar. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'5'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

% Rise

The percentage of final value used with the **Rise time** on page 14-116.

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `80` for Check Linear Step Response Characteristics block

Positive scalar, less than (100 – % settling on page 14-118).

- To assert that the percent rise is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the percent rise from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit**. Specify the new value in **% Rise**. You must click **Update Block** before simulating the model.

Parameter: `PercentRise`
Type: character vector
Value: `[] | 80 |` positive scalar between `0` and `100`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'80'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Settling time

The time, in seconds, taken for the step response to settle within a specified range around the final value. This settling range is defined as the final value plus or minus the percentage of the final value, specified in **% Settling** on page 14-118.

Default:

- `[]` for Linear Step Response Plot block
- `7` for Check Linear Step Response Characteristics block

Finite positive real scalar, greater than rise time on page 14-116.

- To assert that the settling time is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.

- To modify the settling time from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds > Edit**. Specify the new value in **Settling time**. You must click **Update Block** before simulating the model.

Parameter: `SettlingTime`
Type: character vector
Value: `[] | 7 |` positive finite real scalar. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'7'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

% Settling

The percentage of the final value that defines the settling range of the **Settling time** on page 14-117.

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `1` for Check Linear Step Response Characteristics block

Real number, less than (100 – % rise on page 14-116) and less than % overshoot on page 14-119.

- To assert that the percent settling is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the percent settling from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds > Edit**. Specify the new value in **% Settling**. You must click **Update Block** before simulating the model.

Parameter: `PercentSettling`
Type: character vector
Value: `[]` | `1` | real value between `0` and `100`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

% Overshoot

The amount by which the step response can exceed the final value, specified as a percentage.

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `10` for Check Linear Step Response Characteristics block

Real number, greater than % settling on page 14-118.

- To assert that the percent overshoot is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the percent overshoot from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds > Edit**. Specify the new value in **% Overshoot**. You must click **Update Block** before simulating the model.

Parameter: `PercentOvershoot`
Type: character vector

Value: `[]` | `10` | real value between `0` and `100`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'10'` for Check Linear Step Response Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

% Undershoot

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

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `1` for Check Linear Step Response Characteristics block

Real number.

- To assert that the percent undershoot is satisfied, select both **Include step response bound in assertion** on page 14-113 and **Enable assertion** on page 14-124.
- To modify the percent undershoot from the plot window, drag the corresponding bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit**. Specify the new value in **% Undershoot**. You must click **Update Block** before simulating the model.

Parameter: `PercentUndershoot`
Type: character vector
Value: `[]` | `1` | real value between `0` and `100`. Must be specified inside single quotes (`''`).
Default: `'1'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-122.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-121 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

☑ On

Save the operating points.

☐ Off

Do not save the operating points.

Save data to workspace on page 14-121 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

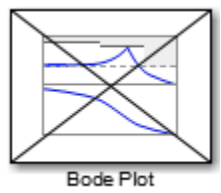
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-125.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-126.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

 On

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

 Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-124 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

 On

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

 Off

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

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

Enable assertion on page 14-124 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

 On

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

 Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

 On

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

 Off

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

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

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

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

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

See Also

Check Linear Step Response Characteristics

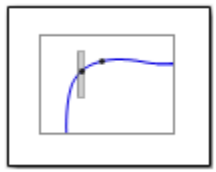
Tutorials

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-77
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-110
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-117
- “Plotting Linear System Characteristics of a Chemical Reactor”

Introduced in R2010b

Nichols Plot

Nichols plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Nichols Characteristics block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a nonlinear Simulink model and plot the linear system on a Nichols plot.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the open-loop gain and phase of the linear system.

The Simulink model can be continuous- or discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

You can specify multiple open- and closed-loop gain and phase bounds and view them on the Nichols plot. You can also check that the bounds are satisfied during simulation:

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

- Evaluate a MATLAB expression.
- Stop the simulation and bring that block into focus.

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

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

You can add multiple Nichols Plot blocks to compute and plot the gains and phases of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Nichols Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/ outputs" on page 14-134. • "Click a signal in the model to select it" on page 14-137.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-140. • "Snapshot times" on page 14-141. • "Trigger type" on page 14-141.

Task		Parameters
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-142. • "Use exact delays" on page 14-144. • "Linear system sample time" on page 14-144. • "Sample time rate conversion method" on page 14-145. • "Prewarp frequency (rad/s)" on page 14-147.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-148. • "Use bus signal names" on page 14-149.
Plot the linear system.		**Show Plot** on page 14-167
Specify the feedback sign for closed-loop gain and phase margins.		"Feedback sign" on page 14-159 in **Bounds** tab.

Task	Parameters
(Optional) Specify bounds on gains and phases of the linear system for assertion.	In **Bounds** tab: • Include gain and phase margins in assertion on page 14-150. • Include closed-loop peak gain in assertion on page 14-153. • Include open-loop gain-phase bound in assertion on page 14-155.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-163. • "Simulation callback when assertion fails (optional)" on page 14-164. • "Stop simulation when assertion fails" on page 14-165. • "Output assertion signal" on page 14-165.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-160 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-166.

Linearization inputs/outputs

Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

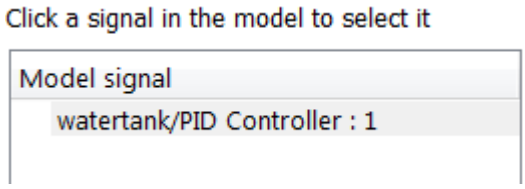
If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click  .

   The dialog box expands to display a **Click a signal in the model to select it** on page 14-137 area and a new  button.

2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select individual elements.

   Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

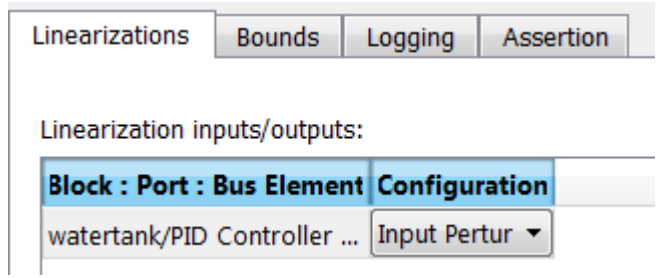
   To modify the filtering options, click  . To hide the filtering options, click  .

   #### Filtering Options

   - “Enable regular expression” on page 14-138
   - “Show filtered results as a flat list” on page 14-139

   ---

4. Click  to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click  .

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click  .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration

Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click  .

When this option appears, you also see the following changes:

- A new « button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-134.

- + changes to −.

  Use − to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

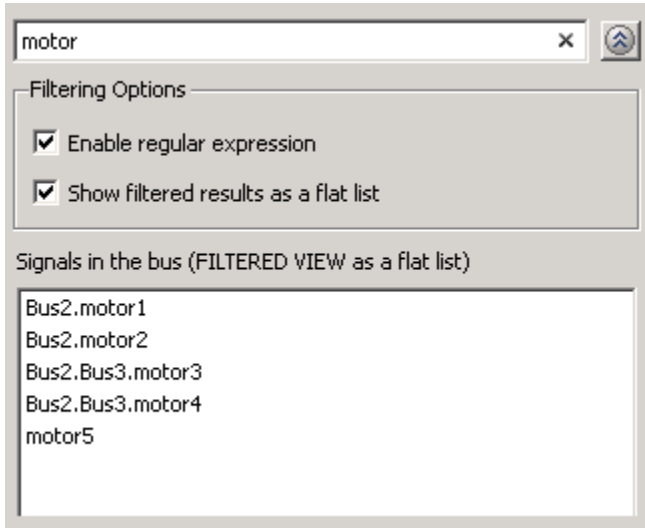
☐ Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

 On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

 Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-141.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-141.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`

Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-140 enables this parameter.

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-140 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising' | 'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

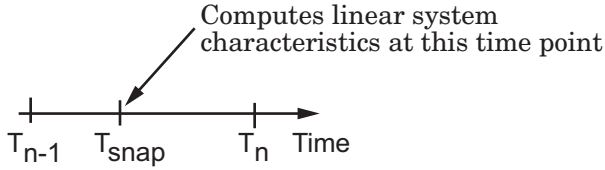
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

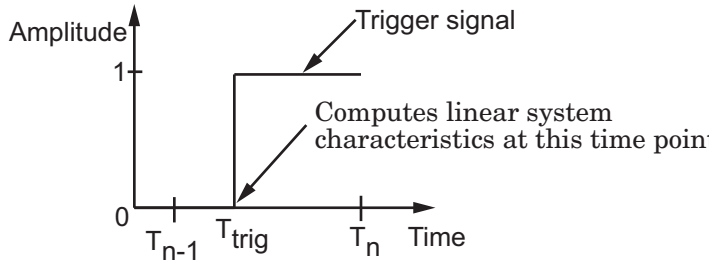
- The exact snapshot times, specified in **Snapshot times** on page 14-141.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-141.

As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

☐ Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

 On

Return a linear model with exact delay representations.

 Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-145.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-144 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-147.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-145 enables this parameter.

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

 On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

 Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

 On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Include gain and phase margins in assertion

Check that the gain and phase margins are greater than the values specified in **Gain margin (dB) >** on page 14-151 and **Phase margin (deg) >** on page 14-152, during simulation. The software displays a warning if the gain or phase margin is less than or equal to the specified value.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the margins.

This parameter is used for assertion only if **Enable assertion** on page 14-163 in the **Assertion** tab is selected.

You can specify multiple gain and phase margin bounds on the linear system. The bounds also appear on the Nichols plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default:

- Off for Nichols Plot block.
- On for Check Nichols Characteristics block.

 On

Check that the gain and phase margins satisfy the specified values, during simulation.

 Off

Do not check that the gain and phase margins satisfy the specified values, during simulation.

- Clearing this parameter disables the gain and phase margin bounds and the software stops checking that the gain and phase margins satisfy the bounds during simulation. The bounds are also greyed out on the plot.
- To only view the gain and phase margin on the plot, clear **Enable assertion**.

Parameter: `EnableMargins`

Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Nichols Plot block, `'on'` for Check Nichols Characteristics block

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Gain margin (dB) >

Gain margin, in decibels.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the gain margin.

Default:
`[]` for Nichols Plot block.
`20` for Check Nichols Characteristics block.

- Positive finite number for one bound.
- Cell array of positive finite numbers for multiple bounds.

- To assert that the gain margin is satisfied, select both **Include gain and phase margins in assertion** on page 14-150 and **Enable assertion** on page 14-163.
- You can add or modify gain margins from the plot window:
  - To add new gain margin, right-click the plot, and select **Bounds** > **New Bound**. Select `Gain margin` in **Design requirement type**, and specify the margin in **Gain margin**.
  - To modify the gain margin, drag the segment. Alternatively, right-click the plot, and select **Bounds** > **Edit Bound**. Specify the new gain margin in **Gain margin >**.

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

Parameter: `GainMargin`
Type: character vector

Value: `[]` | `20` | positive finite value. Must be specified inside single quotes (`''`).
Default: `'[]'` for Nichols Plot block, `'20'` for Check Nichols Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Phase margin (deg) >

Phase margin, in degrees.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the phase margin.

`[]` for Nichols Plot block.
`30` for Check Nichols Characteristics block.

- Positive finite number for one bound.
- Cell array of positive finite numbers for multiple bounds.

- To assert that the phase margin is satisfied, select both **Include gain and phase margins in assertion** on page 14-150 and **Enable assertion** on page 14-163.
- You can add or modify phase margins from the plot window:
  - To add new phase margin, right-click the plot, and select **Bounds** > **New Bound**. Select `Phase margin` in **Design requirement type**, and specify the margin in **Phase margin**.
  - To modify the phase margin, drag the segment. Alternatively, right-click the bound, and select **Bounds** > **Edit Bound**. Specify the new phase margin in **Phase margin >**.

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

Parameter: `PhaseMargin`
Type: character vector
Value: `[]` | `30` | positive finite value. Must be specified inside single quotes (`''`).
Default: `'[]'` for Nichols Plot block, `'30'` for Check Nichols Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include closed-loop peak gain in assertion

Check that the closed-loop peak gain is less than the value specified in **Closed-loop peak gain (dB) <** on page 14-154, during simulation. The software displays a warning if the closed-loop peak gain is greater than or equal to the specified value.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the closed-loop peak gain.

This parameter is used for assertion only if **Enable assertion** on page 14-163 in the **Assertion** tab is selected.

You can specify multiple closed-loop peak gain bounds on the linear system. The bound also appear on the Nichols plot as an m-circle. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default: Off

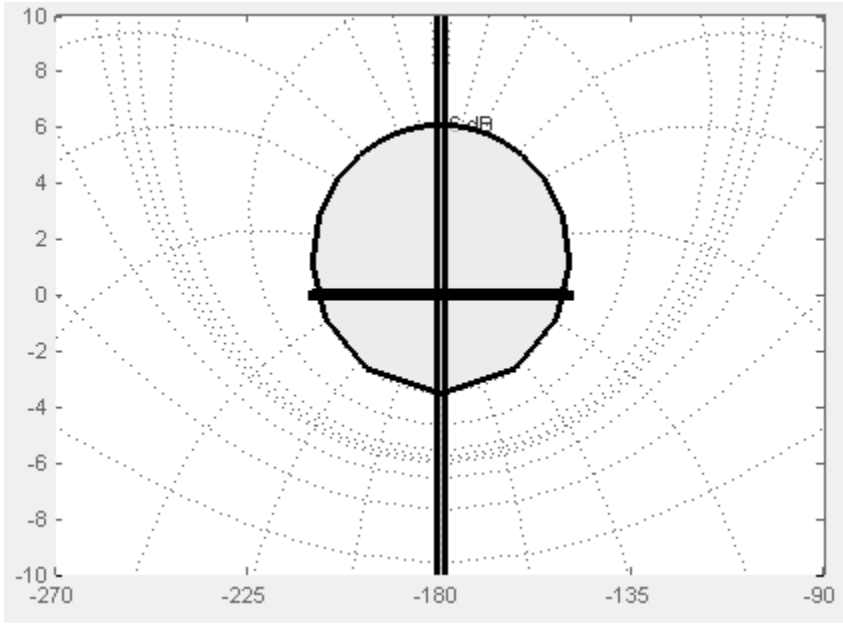
 On

Check that the closed-loop peak gain satisfies the specified value, during simulation.

 Off

Do not check that the closed-loop peak gain satisfies the specified value, during simulation.

- Clearing this parameter disables the closed-loop peak gain bound and the software stops checking that the peak gain satisfies the bounds during simulation. The bounds are greyed out on the plot.



- To only view the closed-loop peak gain on the plot, clear **Enable assertion**.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Closed-loop peak gain (dB) <

Closed-loop peak gain, in decibels.

By default, negative feedback, specified in **Feedback sign** on page 14-83, is used to compute the margins.

Default `[]`

- Positive or negative finite number for one bound.
- Cell array of positive or negative finite numbers for multiple bounds.

- To assert that the gain margin is satisfied, select both **Include closed-loop peak gain in assertion** on page 14-153 and **Enable assertion** on page 14-163.
- You can add or modify closed-loop peak gains from the plot window:
  - To add the closed-loop peak gain, right-click the plot, and select **Bounds** > **New Bound**. Select `Closed-Loop peak gain` in **Design requirement type**, and specify the gain in **Closed-Loop peak gain <**.
  - To modify the closed-loop peak gain, drag the segment. Alternatively, right-click the bound, and select **Bounds** > **Edit Bound**. Specify the new closed-loop peak gain in **Closed-Loop peak gain <**.

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

Parameter: `CLPeakGain`
Type: character vector
Value: `[]` | positive or negative number | cell array of positive or negative numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include open-loop gain-phase bound in assertion

Check that the Nichols response satisfies open-loop gain and phase bounds, specified in **Open-loop phases (deg)** on page 14-157 and **Open-loop gains (dB)** on page 14-158, during simulation. The software displays a warning if the Nichols response violates the bounds.

This parameter is used for assertion only if **Enable assertion** on page 14-163 in the **Assertion** tab is selected.

You can specify multiple gain and phase bounds on the linear systems computed during simulation. The bounds also appear on the Nichols plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default: Off

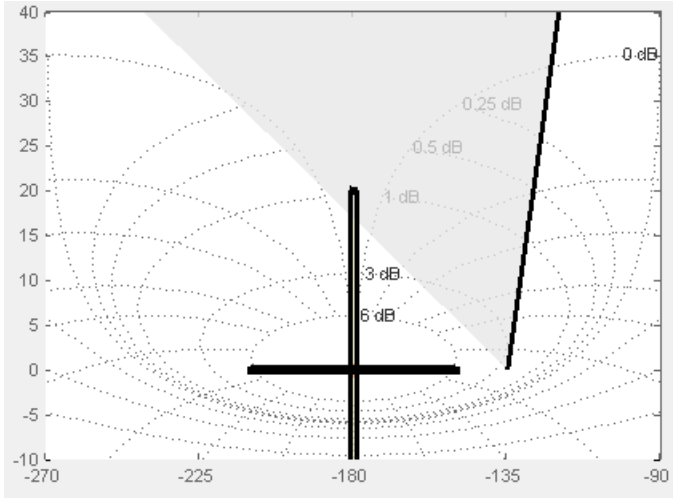
☑ On

Check if the Nichols response satisfies the specified open-loop gain and phase bounds, during simulation.

☐ Off

Do not check if the Nichols response satisfies the specified open-loop gain and phase bounds, during simulation.

- Clearing this parameter disables the gain-phase bound and the software stops checking that the gain and phase satisfy the bound during simulation. The bound segments are also greyed out on the plot.



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

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

Open-loop phases (deg)

Open-loop phases, in degrees.

Specify the corresponding open-loop gains in **Open-loop gains (dB)** on page 14-158.

Default: `[]`

Must be specified as start and end phases:

- Positive or negative finite numbers for a single bound with one edge
- Matrix of positive or negative finite numbers , for a single bound with multiple edges
- Cell array of matrices with finite numbers for multiple bounds

- To assert that the open-loop gains and phases are satisfied, select both **Include open-loop gain-phase bound in assertion** on page 14-155 and **Enable assertion** on page 14-163.
- You can add or modify open-loop phases from the plot window:
  - To add a new phases, right-click the plot, and select **Bounds > New Bound**. Select `Gain-Phase requirement` in **Design requirement type**, and specify the phases in the **Open-Loop phase** column. Specify the corresponding gains in the **Open-Loop gain** column.
  - To modify the phases, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bounds**. Specify the new phases in the **Open-Loop phase** column.

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

Parameter: `OLPhases`
Type: character vector
Value: `[]` | positive or negative finite numbers | matrix of positive or negative finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Open-loop gains (dB)

Open-loop gains, in decibels.

Specify the corresponding open-loop phases in **Open-loop phases (deg)** on page 14-157.

Default: `[]`

Must be specified as start and end gains:

- Positive or negative number for a single bound with one edge
- Matrix of positive or negative finite numbers for a single bound with multiple edges
- Cell array of matrices with finite numbers for multiple bounds

- To assert that the open-loop gains are satisfied, select both **Include open-loop gain-phase bound in assertion** on page 14-155 and **Enable assertion** on page 14-163.
- You can add or modify open-loop gains from the plot window:
  - To add a new gains, right-click the plot, and select **Bounds > New Bound**. Select `Gain-Phase requirement` in **Design requirement type**, and specify the gains in the **Open-Loop phase** column. Specify the phases in the **Open-Loop phase** column.
  - To modify the gains, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bounds**. Specify the new gains in the **Open-Loop gain** column.

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

Parameter: `OLGains`
Type: character vector
Value: `[]` | positive or negative number | matrix of positive or negative finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Feedback sign

Feedback sign to determine the closed-loop gain and phase characteristics of the linear system, computed during simulation.

To determine the feedback sign, check if the path defined by the linearization inputs and outputs include the feedback Sum block:

- If the path includes the Sum block, specify positive feedback.
- If the path does not include the Sum block, specify the same feedback sign as the Sum block.

For example, in the aircraft model, the Check Gain and Phase Margins block includes the negative sign in the summation block. Therefore, the **Feedback sign** is positive.

Default: `negative feedback`

`negative feedback`

Use when the path defined by the linearization inputs/outputs *does not include* the Sum block and the Sum block feedback sign is −.

`positive feedback`

Use when:

- The path defined by the linearization inputs/outputs *includes* the Sum block.
- The path defined by the linearization inputs/outputs *does not include* the Sum block and the Sum block feedback sign is +.

Parameter: `FeedbackSign`
Type: character vector
Value: `'-1' | '+1'`
Default: `'-1'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-161.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-160 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

 On

Save the operating points.

 Off

Do not save the operating points.

Save data to workspace on page 14-160 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

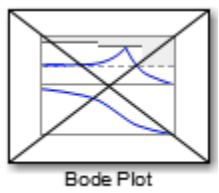
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-164.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-165.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

☑ On

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

☐ Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-163 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

 On

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

 Off

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

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

Enable assertion on page 14-163 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

 On

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

 Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

 On

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

 Off

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

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

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

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

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

See Also

Check Nichols Characteristics

Tutorials

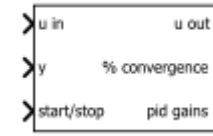
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-77
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-110
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-117
- “Plotting Linear System Characteristics of a Chemical Reactor”

Introduced in R2010b

Online PID Tuner

Automatically tune PID gains based on plant frequency responses estimated from open-loop simulation

Library: Simulink Control Design



Description

The Online PID Tuner block lets you tune a PID controller in real time against a physical plant. The block can tune a PID controller to achieve a specified bandwidth and phase margin without a parametric plant model or an initial controller design. If you have a code-generation product such as Simulink Coder, you can generate code that implements the tuning algorithm on hardware, letting you tune in real time with or without Simulink in the loop.

If you have a plant model in Simulink, you can also use the block to obtain an initial PID design. Doing so lets you preview plant response and adjust the settings for PID autotuning before tuning the controller in real time.

To achieve model-free tuning, the Online PID Tuner block:

1. Injects a test signal into the plant at the nominal operating point to collect plant input-output data and estimate frequency response in real time. The test signal is a combination of sine and step perturbation signals added on top of the nominal plant input measured when the experiment starts. If the plant is part of a feedback loop, the block opens the loop during the experiment.
2. At the end of the experiment, tunes PID controller parameters based on estimated plant frequency responses near the open-loop bandwidth.
3. Updates a PID Controller block or a custom PID controller with the tuned parameters, allowing you to validate closed-loop performance in real time.

Because the block performs an open-loop estimation experiment, do not use this block with an unstable plant or a plant with multiple integrators.

The block supports code generation with Simulink Coder, Embedded Coder®, and Simulink PLC Coder™. It does not support code generation with HDL Coder™.

For more information about using the Online PID Tuner block, see “PID Autotuning in Real Time” on page 7-6.

Ports

Input

`u in` — Signal from PID controller or actuator

scalar

Insert the Online PID Tuner block into your system such that this port accepts a control signal from a source. Typically, this port accepts the signal from the PID controller in your system.

Data Types: `single` | `double`

`y` — Plant output

scalar

Connect this port to the plant output.

Data Types: `single` | `double`

`start/stop` — Start and stop the online tuning experiment

scalar

To start and stop the online tuning process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts. When the experiment is running, the block opens the loop between `u in` and `u out` and injects test signals at `u out`.
- Positive to negative or zero, the experiment stops. When the experiment is not running, the block passes signals unchanged from `u in` to `u out`. In this state, the block has no impact on plant or controller behavior.

Typically, you can control the experiment with a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. Some points to consider when configuring the **start/stop** signal include:

- Start the experiment when the plant is at the desired equilibrium operating point. If necessary, you can use the source block connected to **u in** to drive the plant to the operating point.
- Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.
- Avoid stopping the experiment prematurely. You must let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. There are two ways to determine when stop the experiment:
  - Determine the experiment duration in advance. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning.
  - Observe the signal at the `convergence` output, and stop the experiment when the signal stabilizes near 100%.
- When the experiment stops, the Online PID Tuner block computes tuned PID gains and updates the signal at the `pid gains` port.

Data Types: `single` | `double`

`bandwidth` — Target bandwidth for tuning

scalar

Supply a value for the `Target bandwidth (rad/sec)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target bandwidth (rad/sec)`, select **Use external source**.

Data Types: `single` | `double`

`target PM` — Target phase margin for tuning

scalar

Supply a value for the `Target phase margin (degrees)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target phase margin (degrees)`, select **Use external source**.

Data Types: `single` | `double`

`sine Amp` — Amplitudes of injected sinusoidal signals

scalar | vector

Supply a value for the `Sine Amplitudes` parameter. See that parameter for details.

Dependencies

To enable this port, in the Experiment tab, next to `Sine Amplitudes`, select **Use external source**.

Data Types: `single` | `double`

`step Amp` — Amplitude of injected step signal

scalar

Supply a value for the `Step Amplitude` parameter. See that parameter for details.

Dependencies

To enable this port, in the Experiment tab, next to `Step Amplitudes`, select **Use external source**.

Data Types: `single` | `double`

Output

`u out` — Signal for plant input

scalar

Insert the Online PID Tuner block into your system such that this port feeds the input signal to your plant.

- When the experiment is running (`start/stop` positive), the block injects test signals into the plant at this port. If you have any saturation or rate limit protecting the plant, feed the signal from `u out` into it.
- When the experiment is not running (`start/stop` zero or negative), the block passes signals unchanged from `u in` to `u out`.

Data Types: `single` | `double`

`% convergence` — Convergence of FRD estimation during experiment
scalar

When the experiment is running (`start/stop` positive), the block injects test signals at `u out` and measures the plant response at `y`. It uses these signals to estimate the frequency response of the plant at several frequencies around the target bandwidth for tuning. `% convergence` indicates how close to completion the estimation of the plant frequency response is. Typically, this value quickly rises to about 90% after the experiment begins, and then gradually converges to a higher value. Stop the experiment when it levels off near 100%.

Data Types: `single` | `double`

`pid gains` — Tuned PID coefficients
bus

This bus signal contains the tuned PID gains *P*, *I*, *D*, and the filter coefficient *N*. These values correspond to the `P`, `I`, `D`, and `N` parameters in the expressions given in the `Form` parameter. Initially, the values are 0, 0, 0, and 100, respectively. The block updates the values when the experiment ends.

If you have a PID controller associated with the Online PID Tuner block, you can update that controller with these values after the experiment ends. To do so, in the Block tab, click **Update PID Block**.

Data Types: `single` | `double`

`estimated PM` — Estimated phase margin with tuned controller
scalar

This port outputs the estimated phase margin achieved by the tuned controller, in degrees. The block updates this value when the tuning experiment ends. The estimated phase margin is calculated from the angle of $G(j\omega_c)C(j\omega_c)$, where G is the plant, C is the tuned controller, and ω_c is the crossover frequency (bandwidth). The estimated phase

margin might differ from the target phase margin specified by the `Target phase margin (degrees)` parameter. It is an indicator of the robustness and stability achieved by the tuned system.

- Typically, the estimated phase margin is near the target phase margin. In general, the larger the value, the more robust is the tuned system, and the less overshoot there is.
- A negative phase margin indicates that the closed-loop system might be unstable.

Dependencies

To enable this port, in the Tuning tab, select **Output estimated phase margin achieved by tuned controller**.

`frd` — Estimated frequency response

vector

This port outputs the frequency-response data estimated by the experiment. Initially, the value at `frd` is [0,0,0,0]. During the experiment, the block injects signals at four frequencies, [1/3, 1, 3, 10]ω_c, where ω_c is the target bandwidth. When the experiment stops, the block updates `frd` with a vector containing the complex frequency response at each of these frequencies, respectively.

Dependencies

To enable this port, in the Experiment tab, select **Plant frequency responses near bandwidth**.

`dcgain` — Estimated DC gain of plant

scalar

If you select **Estimate DC gain with step signal** in the Experiment tab, the block estimates the DC gain of the plant by injecting a step signal at `u out`. When the experiment stops, the block updates this port with the estimated DC gain value.

Dependencies

To enable this port, in the Experiment tab, select **Plant DC Gain**.

`nominal` — Plant input and output at nominal operating point

vector

This port outputs a vector containing the plant input (`u out`) and plant output (`y`) when the experiment begins. These values are the plant input and output at the nominal operating point at which the block performs the experiment.

Dependencies

To enable this port, in the Experiment tab, select **Plant nominal input and output**.

Parameters

Tuning Tab

`Type` — PID controller actions

`PI` (default) | `PID` | `PIDF` | ...

Specify the type of the PID controller in your system. The controller type indicates what actions are present in the controller. The following controller types are available for online PID tuning:

- `P` — Proportional only
- `I` — Integral only
- `PI` — Proportional and integral
- `PD` — Proportional and derivative
- `PDF` — Proportional and derivative with derivative filter
- `PID` — Proportional, integral, and derivative
- `PIDF` — Proportional, integral, and derivative with derivative filter

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller type matches.

Tunable: Yes

Programmatic Use

Block Parameter: `PIDType`
Type: character vector
Values: `'P'` | `'I'` | `'PI'` | `'PD'` | `'PDF'` | `'PID'` | `'PIDF'`
Default: `'PI'`

`Form` — PID controller form

`Parallel` (default) | `Ideal`

Specify the controller form. The controller form determines the interpretation of the PID coefficients *P*, *I*, *D*, and *N*.

- `Parallel` — In `Parallel` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P + F_i(z)I + \frac{D}{N + F_d(z)},$$

  where $F_i(z)$ and $F_d(z)$ are the integrator and filter formulas (see `Integrator method` and `Filter method`). The transfer function of a continuous-time parallel-form PIDF controller is:

  $$C = P + \frac{I}{s} + \frac{Ds}{Ns + 1}.$$

  Other controller actions amount to setting *P*, *I*, or *D* to zero.

- `Ideal` — In `Ideal` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P\left(1 + \frac{F_i(z)}{I} + \frac{D}{D/N + F_d(z)}\right).$$

  The transfer function of a continuous-time ideal-form PIDF controller is:

  $$C = P\left(1 + \frac{1}{Is} + \frac{Ds}{Ds/N + 1}\right).$$

  Other controller actions amount to setting *D* to zero or setting, *I* to `Inf`. (In ideal form, the controller must have proportional action.)

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller form matches.

Tunable: Yes

Programmatic Use

Block Parameter: `PIDForm`
Type: character vector

Values: `'Parallel' | 'Ideal'`
Default: `'Parallel'`

`Time Domain` — PID controller time domain

discrete-time (default) | continuous-time

Specify whether your PID controller is a discrete-time or continuous-time controller.

- For discrete time, you must specify the sample time of your PID controller using the **Controller sample time (sec)** parameter.
- For continuous time, you must also specify a sample time for the PID autotuning experiment using the **Experiment sample time (sec)** parameter.

Tunable: No

Programmatic Use

Block Parameter: `TimeDomain`
Type: character vector
Values: `'discrete-time' | 'continuous-time'`
Default: `'discrete-time'`

`Controller sample time (sec)` — Sample time of PID controller

0.1 (default) | positive scalar | –1

Specify the sample time of your PID controller in seconds. This value also sets the sample time for the experiment performed by the block.

To perform PID tuning, the Online PID Tuner measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s ω_c is the controller sample time that you specify with the `Controller sample time (sec)` parameter.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller sample time matches.

Tip If you want to run the deployed block with different sample times in your application, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time. If you do not plan to change the sample time after deployment, specify a fixed and finite sample time.

Tunable: No

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time`.

Programmatic Use

Block Parameter: `DiscreteTs`
Type: scalar
Value positive scalar | –1
Default: 0.1

`Experiment sample time (sec)` — Sample time for experiment

0.02 (default) | positive scalar

Even when you tune a continuous-time controller, you must specify a sample time for the experiment performed by the block. In general, continuous-time controller tuning is not recommended for PID autotuning against a physical plant. If you want to tune in continuous time against a Simulink model of the plant, it is recommended that you use a fast experiment sample time, such as $0.02/\omega_c$.

Tunable: No

Dependencies

This parameter is enabled when the **Time Domain** is `continuous-time`.

Programmatic Use

Block Parameter: `ContinuousTs`
Type: positive scalar
Default: 0.02

`Integrator method` — Discrete integration formula for integrator term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the integrator term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + F_i(z)\,I + \frac{D}{N + F_d(z)},$$

in parallel form, or

$$C = P\left(1 + \frac{F_i(z)}{I} + \frac{D}{D/N + F_d(z)}\right).$$

in ideal form. For a controller sample time T_s, the `Integrator method` parameter determines the formula F_i as follows:

Integrator method	F_i
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the integrator method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes integral action.

Programmatic Use

Block Parameter: `IntegratorFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Filter method` — Discrete integration formula for derivative filter term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the derivative filter term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + F_i(z)\,I + \frac{D}{N + F_d(z)},$$

in parallel form, or

$$C = P\left(1 + \frac{F_i(z)}{I} + \frac{D}{D/N + F_d(z)}\right).$$

in ideal form. For a controller sample time T_s, the `Filter method` parameter determines the formula F_d as follows:

Filter method	F_d
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the filter method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes derivative action.

Programmatic Use

Block Parameter: `FilterFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Target bandwidth (rad/sec)` — Target crossover frequency of open-loop response

1 (default) | positive scalar

The target bandwidth is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ, a good guess for the target bandwidth is $2/\tau$.

To perform PID tuning, the Online PID Tuner block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s is the controller sample time that you specify with the **Controller sample time (sec)** parameter. Because of this condition, the fastest rise time you can enforce for tuning is about $1.67T_s$. If this rise time does not meet your design goals, consider reducing T_s.

To provide the target bandwidth via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `Bandwidth`
Type: positive scalar
Default: `1`

`Target phase margin (degrees)` — Target minimum phase margin of open-loop response

60 (default) | scalar in range 0–90

Specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

To provide the target phase margin via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `TargetPM`

Type: scalar
Values: 0–90
Default: 60

Experiment Tab

`Sine Amplitudes` — Amplitude of sinusoidal perturbations

1 (default) | scalar | vector of length 4

During the tuning experiment, the Online PID Tuner block injects a sinusoidal signal into the plant at four frequencies, $[1/3,1,3,10]\omega_c$, where ω_c is the target bandwidth for tuning. Use **Sine Amplitudes** to specify the amplitude of each of these injected signals. Specify a:

- Scalar value to inject the same amplitude at each frequency.
- Vector of length 4 to specify a different amplitude at each of $[1/3,1,3,10]\omega_c$, respectively.

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the four frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that there is a sharp decay in response over the range of frequencies, consider decreasing the amplitude of $(1/3)\omega_c$ and increasing the amplitude of $10\omega_c$. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level.
- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output.

In the experiment, the four sinusoidal signals are superimposed (with the step perturbation, if any). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

- Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
- How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID

controller used in engine-speed control. You have determined that a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

To provide the sine amplitudes via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use
Block Parameter: `AmpStep`
Type: scalar, vector of length 4
Default: 1

`Estimate DC gain with step signal` — Inject step signal into plant
`on` (default) | `off`

When this option is selected, the experiment includes an estimation of the plant DC gain. The block performs this estimation by injecting a step signal into the plant.

Caution If your plant has a single integrator, clear this option. For plants with multiple integrators or unstable poles, do not use the Online PID Tuner block.

Tunable: Yes

Programmatic Use
Block Parameter: `EstimateDCGain`
Type: character vector
Values: `'off' | 'on'`
Default: `'on'`

`Step Amplitude` — Amplitude of step perturbation
1 (default) | scalar

If **Estimate DC gain with step signal** is selected, the block estimates the DC gain by injecting a step signal into the plant. Use this parameter to set the amplitude of the signal. The considerations for choosing a step amplitude are the same as the considerations for specifying **Sine Amplitudes**.

To provide the step amplitude via an input port, select **Use external source**.

Tunable: Yes

Dependencies

This parameter is enabled when **Estimate DC gain with step signal** is selected.

Programmatic Use

Block Parameter: `AmpSine`
Type: scalar
Default: 1

Block Tab

`Decrease memory footprint (external mode only)` — Deploy tuning algorithm only

`off` (default) | `on`

The Online PID Tuner contains two modules, one that performs the real-time frequency-response estimation, and one that uses the resulting estimated response to tune the PID gains. When you run a Simulink model containing the block in the external simulation mode, by default both modules are deployed. You can save memory on the target hardware by deploying the estimation module only (see "PID Autotuning in External Mode" on page 7-14). In this case, the tuning algorithm runs on the Simulink host computer instead of the target hardware. When this option is selected, the deployed algorithm uses about a third as much memory as when the option is cleared.

Additionally, the PID gain calculation demands more computational load than the frequency-response estimation. For fast controller sample times, some hardware might not finish the gain calculation within one execution cycle. Therefore, when using hardware with limited computing power, selecting this option lets you tune a PID controller with a fast sample time.

If you intend to deploy the block and use it in a standalone application without Simulink, do not select this option.

Tunable: No

Programmatic Use

Block Parameter: `DeployTuningModule`
Type: character vector

Values: `'off'` | `'on'`
Default: `'off'`

`Configure block for PLC Coder` — Configure block for code generation with Simulink PLC Coder
off (default) | on

Select this parameter if you are using Simulink PLC Coder to generate code for the Online PID Tuner block. Clear the parameter for code generation with any other MathWorks code-generation product.

Selecting this parameter affects internal block configuration only, for compatibility with Simulink PLC Coder. The parameter has no operative effect on generated code.

`Data Type` — Floating point precision
`double` (default) | `single`

Specify the floating-point precision based on simulation environment or hardware requirements.

Tunable: No

Programmatic Use
Block Parameter: `BlockDataType`
Type: character vector
Values: `'double'` | `'single'`
Default: `'double'`

`Clicking "Update PID Block" writes tuned gains to the PID block connected to "u in" port` — Automatically detect target for writing tuned PID coefficients
`on` (default) | `off`

Under some conditions, the Online PID Tuner block can write tuned gains to a standard or custom PID controller block. To indicate that the target PID controller is the block connected to the `u in` port of the Online PID Tuner block, select this option. To specify a block that is not connected to `u in`, clear this option.

To write tuned gains from the Online PID Tuner to a PID controller anywhere in the model, the target block must be either:

- A PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters you need for the PID type you are using. For example, if you use a custom PI controller, then you only need mask parameters `P` and `I`.

`Specify PID block path` — Target PID controller block for writing tuned coefficients
`[]` (default) | block path

Under some conditions, the Online PID Tuner block can write tuned gains to a standard or custom PID controller block. Use this parameter to specify the path of the target PID controller.

To write tuned gains from the Online PID Tuner to a PID controller anywhere in the model, the target block must be either:

- A PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters you need for the PID type you are using.

Dependencies

This parameter is enabled when **Clicking "Update PID Block" writes tuned gains to the PID block connected to "u in" port** is selected.

`Update PID Block` — Write tuned PID gains to target controller block
button

The Online PID Tuner block does not automatically push the tuned gains to the target PID block. If your PID controller block meets the criteria described in the `Specify PID block path` parameter description, after tuning, click this button to transfer the tuned gains to the block.

You can update the PID block while the simulation is running, including when running in external mode. Doing so is useful for immediately validating tuned PID gains. At any time during simulation, you can change parameters, start the experiment again, and push the new tuned gains to the PID block. You can then continue to run the model and observe the behavior of your plant.

`Export to MATLAB` — Send experiment and tuning results to MATLAB workspace button

When you click this button, the Online PID Tuner block creates a structure in the MATLAB workspace containing the experiment and tuning results. This structure, `OnlinePIDTuningResult`, contains the following fields:

- `P`, `I`, `D`, `N` — Tuned PID gains. The structure contains whichever of these fields are necessary for the controller type you are tuning. For instance, if you are tuning a PI controller, the structure contains `P` and `I`, but not `D` and `N`.
- `TargetBandwidth` — The value you specified in the **Target bandwidth (rad/sec)** parameter of the block.
- `TargetPhaseMargin` — The value you specified in the **Target phase margin (degrees)** parameter of the block.
- `EstimatedPhaseMargin` — Estimated phase margin achieved by the tuned system.
- `Controller` — The tuned PID controller, returned as a `pid` (for parallel form) or `pidstd` (for ideal form) model object.
- `Plant` — The estimated plant, returned as an `frd` model object. This `frd` contains the response data obtained at the four frequencies [1/3, 1, 3, 10]ω_c.
- `PlantNominal` — The plant input and output at the nominal operating point when the experiment begins, specified as a structure having fields `u` (input) and `y` (output).
- `PlantDCGain` — The estimated DC gain of the system in absolute units, if **Estimate DC gain with step signal** is selected during tuning.

You can export to the MATLAB workspace while the simulation is running, including when running in external mode.

See Also

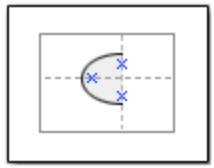
Topics

“PID Autotuning Basics” on page 7-2
“PID Autotuning in Real Time” on page 7-6
“PID Autotuning in External Mode” on page 7-14

Introduced in R2017b

Pole-Zero Plot

Pole-zero plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Pole-Zero Characteristics block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a Simulink model and plot the poles and zeros on a pole-zero map.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the poles and zeros of the linear system.

The Simulink model can be continuous- or discrete-time or multirate and can have time delays. Because you can specify only one linearization input/output pair in this block, the linear system is Single-Input Single-Output (SISO).

You can specify multiple bounds that approximate second-order characteristics on the pole locations and view them on the plot. You can also check that the bounds are satisfied during simulation:

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

- Evaluate a MATLAB expression.
- Stop the simulation and bring that block into focus.

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

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

You can add multiple Pole-Zero Plot blocks to compute and plot the poles and zeros of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Pole-Zero Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/ outputs" on page 14-193. • "Click a signal in the model to select it" on page 14-195.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-198. • "Snapshot times" on page 14-199. • "Trigger type" on page 14-199.

Task		Parameters
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-200. • "Use exact delays" on page 14-202. • "Linear system sample time" on page 14-202. • "Sample time rate conversion method" on page 14-203. • "Prewarp frequency (rad/s)" on page 14-205.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-206. • "Use bus signal names" on page 14-207.
Plot the linear system.		**Show Plot** on page 14-226

Task	Parameters
(Optional) Specify bounds on pole-zero for assertion.	In **Bounds** tab: • Include settling time bound in assertion on page 14-208. • Include percent overshoot bound in assertion on page 14-210. • Include damping ratio bound in assertion on page 14-213. • Include natural frequency bound in assertion on page 14-216.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-221. • "Simulation callback when assertion fails (optional)" on page 14-223. • "Stop simulation when assertion fails" on page 14-223. • "Output assertion signal" on page 14-224.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-218 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-225.

Linearization inputs/outputs

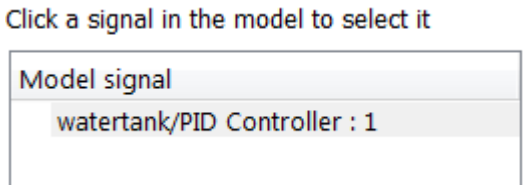
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click  .

   The dialog box expands to display a **Click a signal in the model to select it** on page 14-195 area and a new  button.
2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select individual elements.

Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

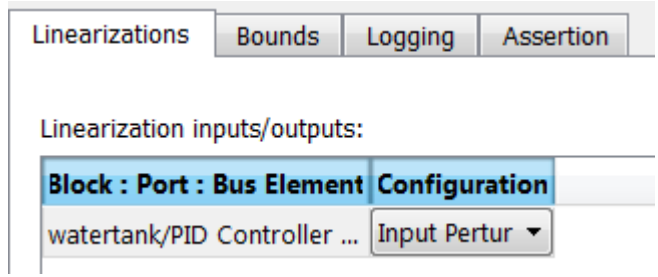
To modify the filtering options, click  . To hide the filtering options, click  .

Filtering Options

- “Enable regular expression” on page 14-196
- “Show filtered results as a flat list” on page 14-197

---

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click  .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration

Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click  .

When this option appears, you also see the following changes:

- A new « button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-193.

- + changes to −.

  Use − to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

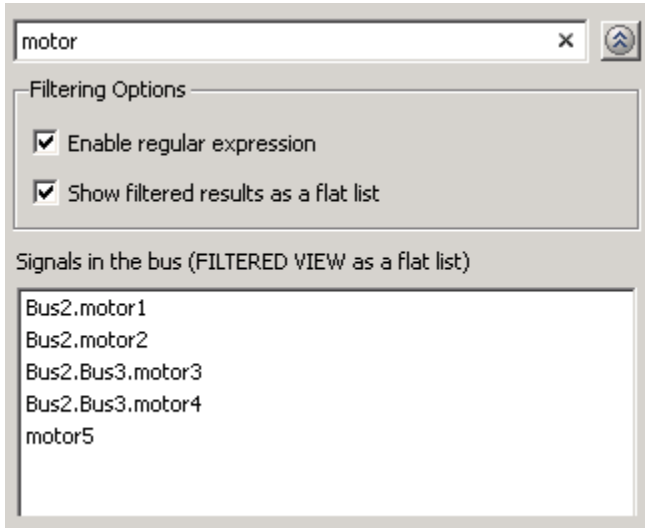
☐ Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

 On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

 Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-199.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-199.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`

Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-198 enables this parameter.

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-198 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising' | 'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

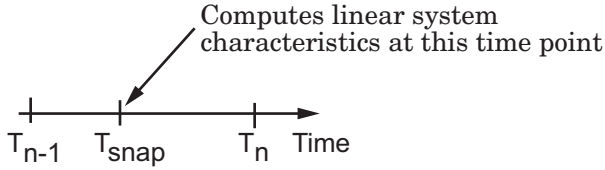
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

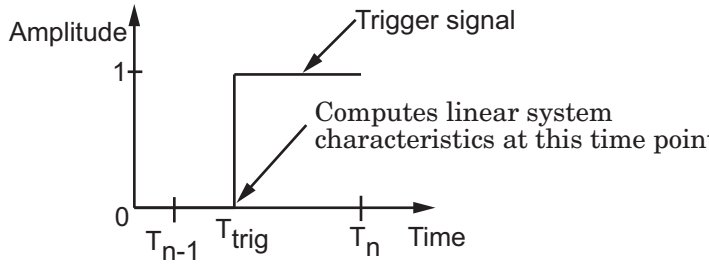
- The exact snapshot times, specified in **Snapshot times** on page 14-199.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-199.

As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

☐ Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

 On

Return a linear model with exact delay representations.

 Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-203.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-202 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-205.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-203 enables this parameter.

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

 On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

 Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

 On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Include settling time bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the settling time, specified in **Settling time (sec) <=** on page 14-209. The software displays a warning if the poles lie outside the region defined by the settling time bound.

This parameter is used for assertion only if **Enable assertion** on page 14-221 in the **Assertion** tab is selected.

You can specify multiple settling time bounds on the linear system. The bounds also appear on the pole-zero plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default:

- Off for Pole-Zero Plot block.
- On for Check Pole-Zero Characteristics block.

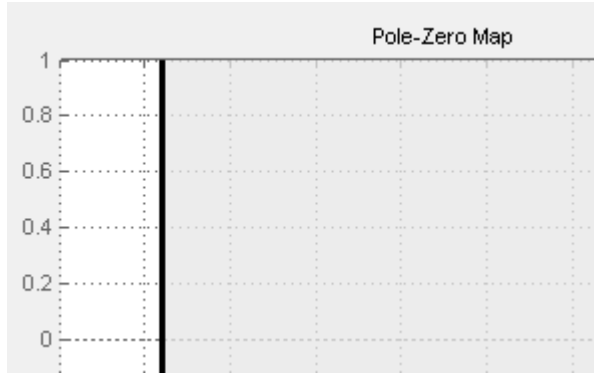
 On

Check that each pole lies in the region defined by the settling time bound, during simulation.

 Off

Do not check that each pole lies in the region defined by the settling time bound, during simulation.

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



- If you also specify other bounds, such as percent overshoot on page 14-210, damping ratio on page 14-213 or natural frequency on page 14-216, but want to exclude the settling time bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Parameter: `EnableSettlingTime`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Pole-Zero Plot block, `'on'` for Check Pole-Zero Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Settling time (sec) <=

Settling time, in seconds, of the second-order system.

Default:
`[]` for Pole-Zero Plot block
`1` for Check Pole-Zero Characteristics block

- Finite positive real scalar for one bound.
- Cell array of finite positive real scalars for multiple bounds.

- To assert that the settling time bounds are satisfied, select both **Include settling time bound in assertion** on page 14-208 and **Enable assertion** on page 14-221.
- You can add or modify settling time bounds from the plot window:
  - To add a new settling time bound, right-click the plot, and select **Bounds** > **New Bound**. Specify the new value in **Settling time**.
  - To modify a settling time bound, drag the corresponding bound segment. Alternatively, right-click the bound and select **Bounds** > **Edit**. Specify the new value in **Settling time (sec) <**.

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

Parameter: `SettlingTime`
Type: character vector
Value: `[] | 1 | finite positive real scalar| cell array of finite positive real scalars`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Pole-Zero Plot block, `'1'` for Check Pole-Zero Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include percent overshoot bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the percent overshoot, specified in **Percent overshoot <=** on page 14-209. The software displays a warning if the poles lie outside the region defined by the percent overshoot bound.

This parameter is used for assertion only if **Enable assertion** on page 14-221 in the **Assertion** tab is selected.

You can specify multiple percent overshoot bounds on the linear system. The bounds also appear on the pole-zero plot. If you clear **Enable assertion**, the bounds are not used for assertion but continues to appear on the plot.

Default:
Off for Pole-Zero Plot block.

On for Check Pole-Zero Characteristics block.

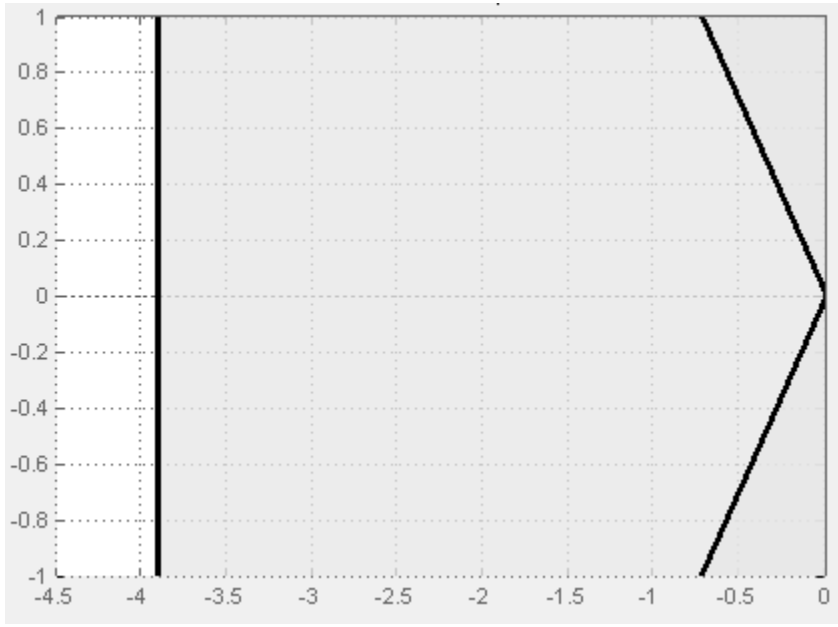
 On

Check that each pole lies in the region defined by the percent overshoot bound, during simulation.

 Off

Do not check that each pole lies in the region defined by the percent overshoot bound, during simulation.

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



- If you specify other bounds, such as settling time on page 14-208, damping ratio on page 14-213 or natural frequency on page 14-216, but want to exclude the percent overshoot bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Parameter: `EnablePercentOvershoot`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Pole-Zero Plot block, `'on'` for Check Pole-Zero Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Percent overshoot <=

Percent overshoot of the second-order system.

Default:
`[]` for Pole-Zero Plot block
`10` for Check Pole-Zero Characteristics block

Minimum: 0

Maximum: 100

- Real scalar for single percent overshoot bound.
- Cell array of real scalars for multiple percent overshoot bounds.

- The percent overshoot *p.o* can be expressed in terms of the damping ratio on page 14-215 ζ, as:

  $p.o. = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$.

- To assert that the percent overshoot bounds are satisfied, select both **Include percent overshoot bound in assertion** on page 14-208 and **Enable assertion** on page 14-221.
- You can add or modify percent overshoot bounds from the plot window:
  - To add a new percent overshoot bound, right-click the plot, and select **Bounds** > **New Bound**. Select `Percent overshoot` in **Design requirement type** and specify the value in **Percent overshoot <**.

- To modify a percent overshoot bound, drag the corresponding bound segment. Alternatively, right-click the bound, and select **Bounds** > **Edit**. Specify the new damping ratio for the corresponding percent overshoot value in **Damping ratio >**.

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

Parameter: `PercentOvershoot`
Type: character vector
Value: `[]` | `10` | `real scalar between 0 and 100` | `cell array of real scalars between 0 and 100`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Pole-Zero Plot block, `'10'` for Check Pole-Zero Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include damping ratio bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the damping ratio, specified in **Damping ratio >=** on page 14-213. The software displays a warning if the poles lie outside the region defined by the damping ratio bound.

This parameter is used for assertion only if **Enable assertion** on page 14-221 in the **Assertion** tab is selected.

You can specify multiple damping ratio bounds on the linear system. The bounds also appear on the pole-zero plot. If you clear **Enable assertion**, the bounds are not used for assertion but continues to appear on the plot.

Default: Off

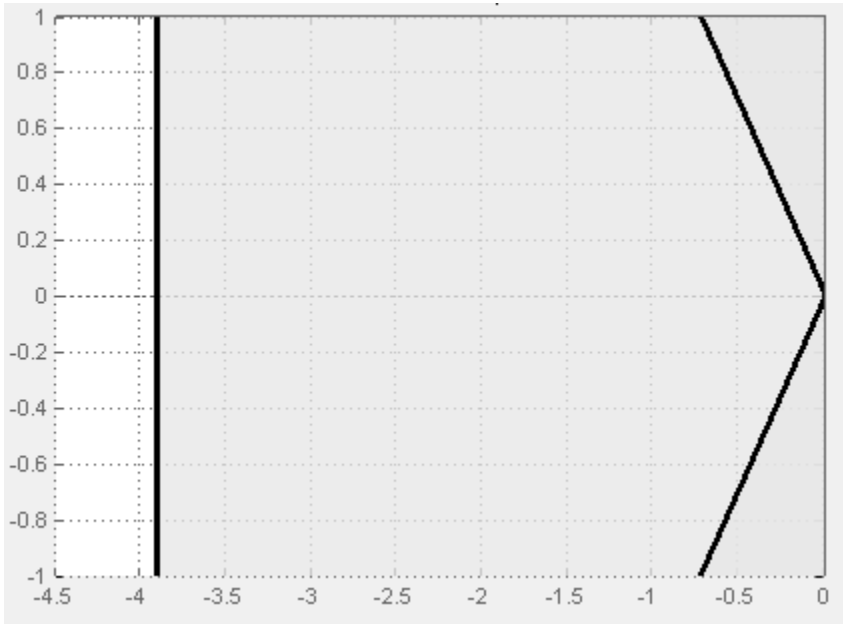
 On

Check that each pole lies in the region defined by the damping ratio bound, during simulation.

 Off

Do not check that each pole lies in the region defined by the damping ratio bound, during simulation.

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



- If you specify other bounds, such as settling time on page 14-208, percent overshoot on page 14-210 or natural frequency on page 14-216, but want to exclude the damping ratio bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Damping ratio >=

Damping ratio of the second-order system.

Default: `[]`

Minimum: 0

Maximum: 1

- Finite positive real scalar for single damping ratio bound.
- Cell array of finite positive real scalars for multiple damping ratio bounds.

- The damping ratio ζ, and percent overshoot *p.o* are related as:
  $p.o. = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$.
- To assert that the damping ratio bounds are satisfied, select both **Include damping ratio bound in assertion** on page 14-213 and **Enable assertion** on page 14-221.
- You can add or modify damping ratio bounds from the plot window:
  - To add a new damping ratio bound, right-click the plot and select **Bounds** > **New Bound**. Select `Damping ratio` in **Design requirement type** and specify the value in **Damping ratio >**.
  - To modify a damping ratio bound, drag the corresponding bound segment or right-click it and select **Bounds** > **Edit**. Specify the new value in **Damping ratio >**.

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

Parameter: `DampingRatio`
Type: character vector
Value: `[]` | `finite positive real scalar between 0 and 1` | `cell array of finite positive real scalars between 0 and 1` . Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include natural frequency bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the natural frequency, specified in **Natural frequency (rad/sec)** on page 14-217. The natural frequency bound can be greater than, less than or equal one or more specific values. The software displays a warning if the pole locations do not satisfy the region defined by the natural frequency bound.

This parameter is used for assertion only if **Enable assertion** on page 14-221 in the **Assertion** tab is selected.

You can specify multiple natural frequency bounds on the linear system. The bounds also appear on the pole-zero plot. If **Enable assertion** is cleared, the bounds are not used for assertion but continue to appear on the plot.

Default: Off

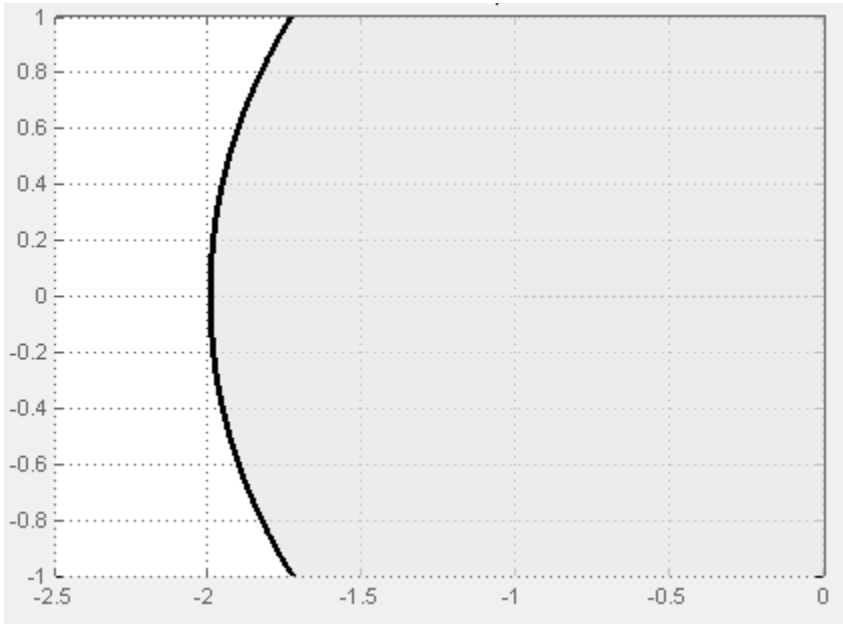
 On

Check that each pole lies in the region defined by the natural frequency bound, during simulation.

 Off

Do not check that each pole lies in the region defined by the natural frequency bound, during simulation.

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



- If you also specify settling time on page 14-208, percent overshoot on page 14-210 or damping ratio on page 14-213 bounds and want to exclude the natural frequency bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

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

Natural frequency (rad/sec)

Natural frequency of the second-order system.

Default: `[]`

- Finite positive real scalar for single natural frequency bound.
- Cell array of finite positive real scalars for multiple natural frequency bounds.

- To assert that the natural frequency bounds are satisfied, select both **Include natural frequency bound in assertion** on page 14-213 and **Enable assertion** on page 14-221.
- You can add or modify natural frequency bounds from the plot window:
  - To add a new natural frequency bound, right-click the plot and select **Bounds** > **New Bound**. Select `Natural frequency` in **Design requirement type** and specify the natural frequency in **Natural frequency**.
  - To modify a natural frequency bound, drag the corresponding bound segment or right-click it and select **Bounds** > **Edit**. Specify the new value in **Natural frequency**.

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

Parameter: `NaturalFrequency`
Type: character vector
Value: `[]` | `positive finite real scalar` | `cell array of positive finite real scalars`. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation** > **Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-220.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see “Export Simulation Data” (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-218 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a

field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

 On

Save the operating points.

 Off

Do not save the operating points.

Save data to workspace on page 14-218 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

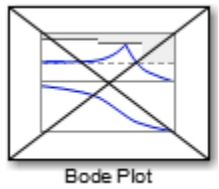
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-223.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-223.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

☑ On

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

☐ Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-221 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

☑ On

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

Off

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

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

Enable assertion on page 14-221 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see "Verify Model Using Simulink Control Design and Simulink Verification Blocks" on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

On

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

Off

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

Parameter: `LaunchViewOnOpen`

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking  or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking  or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.

- Simulating the model by clicking  or selecting **Simulation** > **Start**. This action also linearizes the portion of the model between the specified linearization input and output.
- Adding legend on the linear system characteristic plot by clicking  .

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools** > **Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

- "Design Optimization to Meet Step Response Requirements (GUI)" (Simulink Design Optimization)
- "Design Optimization to Meet Time- and Frequency-Domain Requirements (GUI)" (Simulink Design Optimization)

See Also

Check Pole-Zero Characteristics

Tutorials

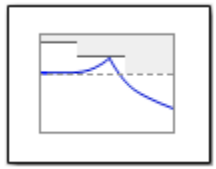
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-77
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-110
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-117

- “Plotting Linear System Characteristics of a Chemical Reactor”

Introduced in R2010b

Singular Value Plot

Singular value plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is same as the Check Singular Value Characteristics block except for different default parameter settings in the **Bounds** tab.

Compute a linear system from a nonlinear Simulink model and plot the linear system on a singular value plot.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs, and plots the singular values of the linear system.

The Simulink model can be continuous- or discrete-time or multirate, and can have time delays. The linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO). For MIMO systems, the plots for all input/output combinations are displayed.

You can specify piecewise-linear frequency-dependent upper and lower singular value bounds and view them on the plot. You can also check that the bounds are satisfied during simulation:

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

- Evaluate a MATLAB expression.
- Stop the simulation and bring that block into focus.

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

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

For MIMO systems, the bounds apply to the singular values of linear systems computed for all input/output combinations.

You can add multiple Singular Value Plot blocks to compute and plot the singular values of various portions of the model.

You can save the linear system as a variable in the MATLAB workspace.

The block does not support code generation and can be used only in `Normal` simulation mode.

Parameters

The following table summarizes the Singular Value Plot block parameters, accessible via the block parameter dialog box.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • "Linearization inputs/ outputs" on page 14-232. • "Click a signal in the model to select it" on page 14-235.

Task		Parameters
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 14-237. • "Snapshot times" on page 14-238. • "Trigger type" on page 14-239.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 14-239. • "Use exact delays" on page 14-241. • "Linear system sample time" on page 14-242. • "Sample time rate conversion method" on page 14-243. • "Prewarp frequency (rad/s)" on page 14-244.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 14-245. • "Use bus signal names" on page 14-246.
Plot the linear system.		**Show Plot** on page 14-262

Task	Parameters
(Optional) Specify bounds on singular values for assertion.	In **Bounds** tab: • Include upper singular value bound in assertion on page 14-247. • Include lower singular value bound in assertion on page 14-251.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 14-258. • "Simulation callback when assertion fails (optional)" on page 14-259. • "Stop simulation when assertion fails" on page 14-260. • "Output assertion signal" on page 14-260.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 14-255 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 14-261.

Linearization inputs/outputs

Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

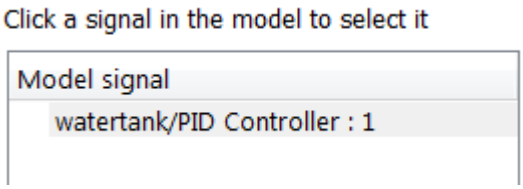
If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click  at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1 Click  .

The dialog box expands to display a **Click a signal in the model to select it** on page 14-235 area and a new  button.

2 Select one or more signals in the Simulink Editor.

The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression (MATLAB).

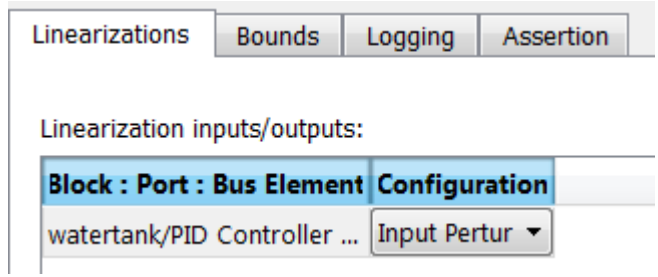
To modify the filtering options, click  . To hide the filtering options, click  .

Filtering Options

- “Enable regular expression” on page 14-235
- “Show filtered results as a flat list” on page 14-236

---

4 Click  to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click  .

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click  .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration — Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

No default

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click  .

When this option appears, you also see the following changes:

- A new  button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 14-232.
-  changes to  .

  Use  to collapse the **Click a signal in the model to select it** area.

No default

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable regular expression

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a

lowercase `t` (and their immediate parents). For details, see "Regular Expressions" (MATLAB).

Default: On

☑ On

Allow use of MATLAB regular expressions for filtering signal names.

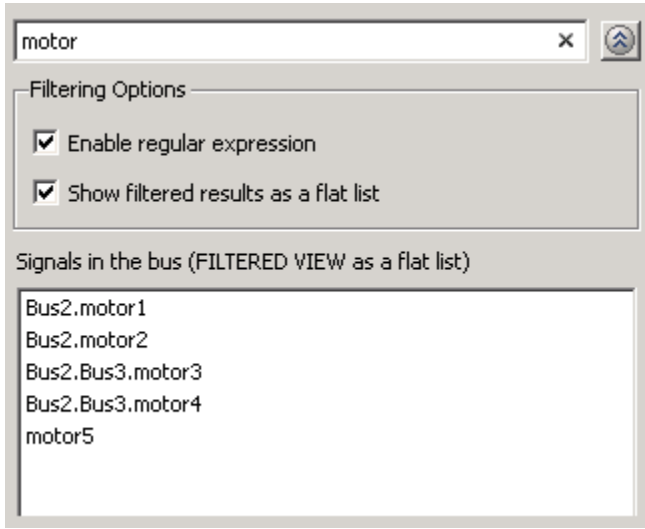
☐ Off

Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Show filtered results as a flat list

Uses a flat list format to display the list of filtered signals, based on the search text in the **Filter by name** edit box. The flat list format uses dot notation to reflect the hierarchy of bus signals. The following is an example of a flat list format for a filtered set of nested bus signals.



Default: Off

On

Display the filtered list of signals using a flat list format, indicating bus hierarchies with dot notation instead of using a tree format.

Off

Display filtered bus hierarchies using a tree format.

Selecting the **Options** button on the right-hand side of the **Filter by name** edit box () enables this parameter.

Linearize on

When to compute the linear system during simulation.

Default: `Simulation snapshots`

`Simulation snapshots`

Specific simulation time, specified in **Snapshot times** on page 14-238.

Use when you:

- Know one or more times when the model is at steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 14-239.

Use when a signal generated during simulation indicates steady-state operating point.

Selecting this option adds a trigger port to the block. Use this port to connect the block to the trigger signal.

For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a fraction of the maximum fuel mass. In this case, model this condition as an external trigger.

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Default: 0

- For a different simulation time, enter the time. Use when you:
  - Want to plot the linear system at a specific time
  - Know the approximate time when the model reaches steady-state operating point
- For multiple simulation times, enter a vector. Use when you want to compute and plot linear systems at multiple times.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

Selecting `Simulation snapshots` in **Linearize on** on page 14-237 enables this parameter.

Parameter: `SnapshotTimes`

Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Trigger type

Trigger type of an external trigger for computing linear system.

Default: `Rising edge`

`Rising edge`

Rising edge of the external trigger signal.

`Falling edge`

Falling edge of the external trigger signal.

Selecting `External trigger` in **Linearize on** on page 14-237 enables this parameter.

Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

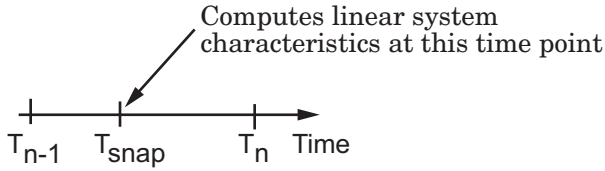
"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

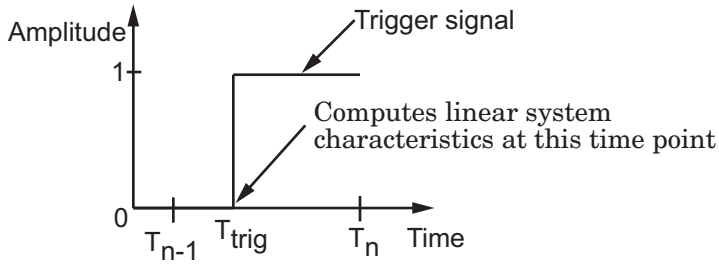
- The exact snapshot times, specified in **Snapshot times** on page 14-238.

As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 14-239.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the time, T_{trig}, when the trigger signal is detected. T_{trig} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



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

Default: On

☑ On

Compute linear system characteristics at the exact snapshot time or exact time when a trigger signal is detected.

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

Off

Compute linear system characteristics at the simulation time steps that the variable-step solver chooses. The software may not compute the linear system at the exact snapshot time or exact time when a trigger signal is detected.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use exact delays

How to represent time delays in your linear model.

Use this option if you have blocks in your model that have time delays.

Default: Off

On

Return a linear model with exact delay representations.

Off

Return a linear model with Padé approximations of delays, as specified in your Transport Delay and Variable Transport Delay blocks.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 14-243.

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multi-rate models), least common multiple of the sample times. For example, if you have a mix of continuous-time and discrete-time blocks with sample times of 0, 0.2 and 0.3, the sample time of the linear model is 0.6.

Positive finite value. Use to compute:

- A discrete-time linear system from a continuous-time system.
- A discrete-time linear system from another discrete-time system with a different sample time

`0`

Use to compute a continuous-time linear system from a discrete-time model.

Parameter: `SampleTime`
Type: character vector
Value: `auto` | Positive finite value | `0`
Default: `auto`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Sample time rate conversion method

Method for converting the sample time of single- or multi-rate models.

This parameter is used only when the value of **Linear system sample time** on page 14-242 is not `auto`.

Default: `Zero-Order Hold`

`Zero-Order Hold`

Zero-order hold, where the control inputs are assumed piecewise constant over the sampling time `Ts`. For more information, see "Zero-Order Hold" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually performs better in time domain.

`Tustin (bilinear)`

Bilinear (Tustin) approximation without frequency prewarping. The software rounds off fractional time delays to the nearest multiple of the sampling time. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain.

`Tustin with Prewarping`

Bilinear (Tustin) approximation with frequency prewarping. Also specify the prewarp frequency in **Prewarp frequency (rad/s)**. For more information, see "Tustin Approximation" (Control System Toolbox) in *Control System Toolbox User's Guide*.

This method usually perform better in the frequency domain. Use this method to ensure matching at frequency region of interest.

`Upsampling when possible, Zero-Order Hold otherwise`

Upsample a discrete-time system when possible and use `Zero-Order Hold` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin otherwise`

Upsample a discrete-time system when possible and use `Tustin (bilinear)` otherwise.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

`Upsampling when possible, Tustin with Prewarping otherwise`

Upsample a discrete-time system when possible and use `Tustin with Prewarping` otherwise. Also, specify the prewarp frequency in **Prewarp frequency (rad/s)**.

You can upsample only when you convert discrete-time system to a new sample time that is an integer-value-times faster than the sampling time of the original system.

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 14-244.

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Default: 10

Positive scalar value, smaller than the Nyquist frequency before and after resampling. A value of `0` corresponds to the standard Tustin method without frequency prewarping.

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** on page 14-243 enables this parameter.

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use full block names

How the state, input and output names appear in the linear system computed during simulation.

The linear system is a state-space object and system states and input/output names appear in following state-space object properties:

Input, Output or State Name	Appears in Which State-Space Object Property
Linearization input name	`InputName`
Linearization output name	`OutputName`
State names	`StateName`

Default: Off

On

Show state and input/output names with their path through the model hierarchy. For example, in the `chemical reactor model`, a state in the `Integrator1` block of the `CSTR` subsystem appears with full path as `scdcstr/CSTR/Integrator1`.

Off

Show only state and input/output names. Use this option when the signal name is unique and you know where the signal is location in your Simulink model. For example, a state in the `Integrator1` block of the `CSTR` subsystem appears as `Integrator1`.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Use bus signal names

How to label signals associated with linearization inputs and outputs on buses, in the linear system computed during simulation (applies only when you select an entire bus as an I/O point).

Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

You cannot use this parameter when your model has mux/bus mixtures.

Default: Off

On

Use the signal names of the individual bus elements.

Bus signal names appear when the input and output are at the output of the following blocks:

- Root-level inport block containing a bus object
- Bus creator block
- Subsystem block whose source traces back to one of the following blocks:
  - Output of a bus creator block
  - Root-level inport block by passing through only virtual or nonvirtual subsystem boundaries

 Off

Use the bus signal channel number.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include upper singular value bound in assertion

Check that the singular values satisfy upper bounds, specified in **Frequencies (rad/sec)** on page 14-249 and **Magnitude (dB)** on page 14-250, during simulation. The software displays a warning during simulation if the singular values violate the upper bound.

This parameter is used for assertion only if **Enable assertion** on page 14-258 in the **Assertion** tab is selected.

You can specify multiple upper singular value bounds on the linear system. The bounds also appear on the singular value plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default:

- Off for Singular Value Plot block.

- On for Check Singular Value Characteristics block.

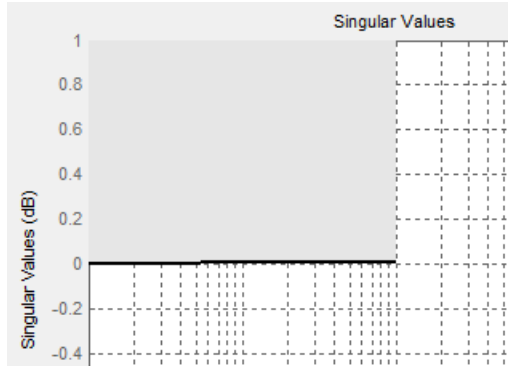

On

Check that the singular value satisfies the specified upper bounds, during simulation.


Off

Do not check that the singular value satisfies the specified upper bounds, during simulation.

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



- If you specify both upper and lower singular value bounds on page 14-251 but want to include only the lower bounds for assertion, clear this parameter.
- To only view the bound on the plot, clear **Enable assertion**.

Parameter: `EnableUpperBound`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'` for Singular Value Plot block, `'on'` for Check Singular Value Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Frequencies (rad/sec)

Frequencies for one or more upper singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 14-250.

Default:
`[]` for Singular Value Plot block
`[0.1 100]` for Check Singular Value Characteristics block

Must be specified as start and end frequencies:

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

  For example, type [0.1 1;1 10] for two edges at frequencies [0.1 1] and [1 10].
- Cell array of matrices with positive finite numbers for multiple bounds.

- To assert that magnitudes that correspond to the frequencies are satisfied, select both **Include upper singular value bound in assertion** on page 14-247 and **Enable assertion** on page 14-258.
- You can add or modify frequencies from the plot window:
  - To add new frequencies, right-click the plot, and select **Bounds** > **New Bound**. Select `Upper gain limit` in **Design requirement type**, and specify the frequencies in the **Frequency** column. Specify the corresponding magnitudes in the **Magnitude** column.
  - To modify the frequencies, drag the bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit Bound**. Specify the new frequencies in the **Frequency** column.

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

Parameter: `UpperBoundFrequencies`
Type: character vector

Value: `[]` | `[0.1 100]` | positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Singular Value Plot block, `'[0.1 100]'` for Check Singular Value Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Magnitudes (dB)

Magnitude values for one or more upper singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 14-249.

Default:
`[]` for Singular Value Plot block
`[0 0]` for Check Singular Value Characteristics block

Must be specified as start and end magnitudes:

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

  For example, type [0 0; 10 10] for two edges at magnitudes [0 0] and [10 10].
- Cell array of matrices with finite numbers for multiple bounds

- To assert that magnitudes are satisfied, select both **Include upper singular value bound in assertion** on page 14-247 and **Enable assertion** on page 14-258.
- You can add or modify magnitudes from the plot window:
  - To add a new magnitude, right-click the plot, and select **Bounds > New Bound**. Select `Upper gain limit` in **Design requirement type**, and specify the magnitude in the **Magnitude** column. Specify the corresponding frequencies in the **Frequency** column.

- To modify the magnitudes, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bound**. Specify the new magnitudes in the **Magnitude** column.

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

Parameter: `UpperBoundMagnitudes`
Type: character vector
Value: `[]` | `[0 0]` | finite number | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Singular Value Plot block, `'[0 0]'` for Check Singular Value Characteristics block.

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Include lower singular value bound in assertion

Check that the singular values satisfy lower bounds, specified in **Frequencies (rad/sec)** on page 14-252 and **Magnitude (dB)** on page 14-254, during simulation. The software displays a warning if the singular values violate the lower bound.

This parameter is used for assertion only if **Enable assertion** on page 14-258 in the **Assertion** tab is selected.

You can specify multiple lower singular value bounds on the linear system. The bounds also appear on the singular value plot. If you clear **Enable assertion**, the bounds are not used for assertion but continue to appear on the plot.

Default: Off

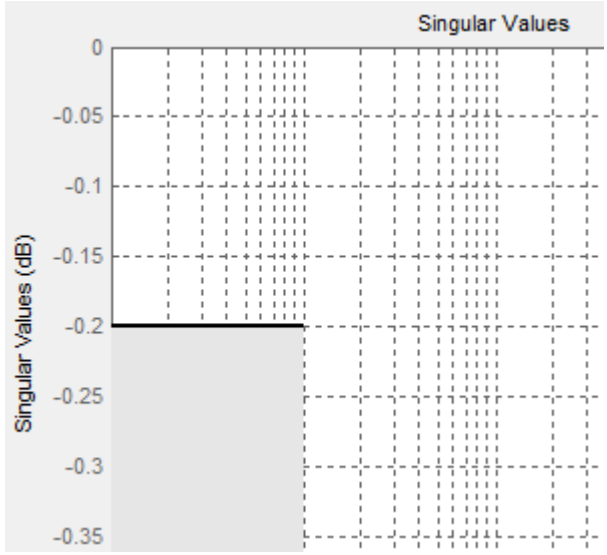
☑ On

Check that the singular value satisfies the specified lower bounds, during simulation.

☐ Off

Do not check that the singular value satisfies the specified lower bounds, during simulation.

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



- If you specify both lower and upper singular value bounds on page 14-247 but want to include only the upper bounds for assertion, clear this parameter.
- To only view the bound on the plot, clear **Enable assertion**.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Frequencies (rad/sec)

Frequencies for one or more lower singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 14-254.

Default `[]`

Must be specified as start and end frequencies:

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

  For example, type [0.01 0.1;0.1 1] to specify two edges with frequencies [0.01 0.1] and [0.1 1].
- Cell array of matrices with positive finite numbers for multiple bounds.

- To assert that magnitude bounds that correspond to the frequencies are satisfied, select both **Include lower singular value bound in assertion** on page 14-251 and **Enable assertion** on page 14-258.
- You can add or modify frequencies from the plot window:
  - To add new frequencies, right-click the plot, and select **Bounds** > **New Bound**. Select `Lower gain limit` in **Design requirement type** and specify the frequencies in the **Frequency** column. Specify the corresponding magnitudes in the **Magnitude** column.
  - To modify the frequencies, drag the bound segment. Alternatively, right-click the segment, and select **Bounds** > **Edit Bound**. Specify the new frequencies in the **Frequency** column.

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

Parameter: `LowerBoundFrequencies`
Type: character vector
Value: `[]` | positive finite number | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Magnitudes (dB)

Magnitude values for one or more lower singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 14-252.

Default `[]`

Must be specified as start and end magnitudes:

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

  For example, type [0 0; 10 10] for two edges with magnitudes [0 0] and [10 10].
- Cell array of matrices with finite numbers for multiple bounds

- To assert that magnitudes are satisfied, select both **Include lower singular value bound in assertion** on page 14-251 and **Enable assertion** on page 14-258.
- You can add or modify magnitudes from the plot window:

  - To add new magnitudes, right-click the plot, and select **Bounds > New Bound**. Select `Lower gain limit` in **Design requirement type**, and specify the magnitudes in the **Magnitude** column. Specify the corresponding frequencies in the **Frequency** column.
  - To modify the magnitudes, drag the bound segment. Alternatively, right-click the segment, and select **Bounds > Edit Bound**. Specify the new magnitudes in the **Magnitude** column.

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

Parameter: `LowerBoundFrequencies`
Type: character vector
Value: `[]` | finite number | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, in the Data Import/Export pane, check **Single simulation output**.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: Off

 On

Save the computed linear system.

 Off

Do not save the computed linear system.

This parameter enables **Variable name** on page 14-256.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Variable name

Name of the data structure that stores one or more linear systems computed during simulation.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data. The

The name must be unique among the variable names used in all data logging model blocks, such as Linear Analysis Plot blocks, Model Verification blocks, Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs.

For more information about data logging in Simulink, see "Export Simulation Data" (Simulink) and the `Simulink.SimulationOutput` reference page.

Default: sys

Character vector.

Save data to workspace on page 14-255 enables this parameter.

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Save operating points for each linearization

When saving linear systems to the workspace for further analysis or control design, also save the operating point corresponding to each linearization. Using this option adds a field named `operatingPoints` to the data structure that stores the saved linear systems.

Default: Off

☑ On

Save the operating points.

☐ Off

Do not save the operating points.

Save data to workspace on page 14-255 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Enable assertion

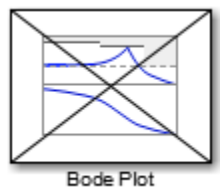
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 14-259.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 14-260.

For the `Linear Analysis Plots` blocks, this parameter has no effect because no bounds are included by default. If you want to use the `Linear Analysis Plots` blocks for assertion, specify and include bounds in the **Bounds** tab.

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



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

Default: On

☑ On

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

☐ Off

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

This parameter enables:

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Simulation callback when assertion fails (optional)

MATLAB expression to execute when assertion fails.

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

No Default

A MATLAB expression.

Enable assertion on page 14-258 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Stop simulation when assertion fails

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

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

Default: Off

 On

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

 Off

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

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

Enable assertion on page 14-258 enables this parameter.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Output assertion signal

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

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

The output signal data type is Boolean only if, in theSimulinkmodel, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

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

Default: Off

 On

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

 Off

Do not output a Boolean signal to indicate assertion status.

- Use this parameter to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 12-25.

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

“Verify Model at Default Simulation Snapshot Time” on page 12-6

Show plot on block open

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

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

For more information on the plot, see **Show Plot** on page 14-36.

Default: Off

 On

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

 Off

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

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

Plot Linear Characteristics of Simulink Models During Simulation on page 2-77

"Verify Model at Default Simulation Snapshot Time" on page 12-6

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

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

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

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

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

Typical tasks that you perform in the plot window include:

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

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the Response Optimization tool by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

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

This button is available only if you have Simulink Design Optimization software installed.

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

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

See Also

Check Singular Value Characteristics

Tutorials

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-77
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-110
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-117
- “Plotting Linear System Characteristics of a Chemical Reactor”

Introduced in R2010b

Trigger-Based Operating Point Snapshot

Generate operating points, linearizations, or both at triggered events

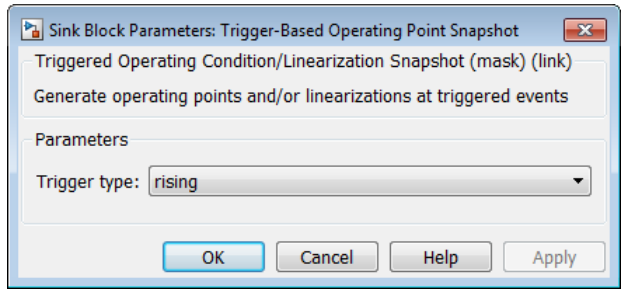
Library

Simulink Control Design

Description

Attach this block to a signal in a model when you want to take a snapshot of the system's operating point at triggered events such as when the signal crosses zero or when the signal sends a function call. You can also perform a linearization at these events. To extract the operating point or perform the linearization, you need to simulate the model using either the `findop` or `linearize` functions. Alternatively, you can interactively export the operating point and linearize the model using the Linear Analysis Tool.

Choose the trigger type in the Block Parameters dialog box, as shown in the following figure.



The possible trigger types are

- `rising`: the signal crosses zero while increasing.
- `falling`: the signal crosses zero while decreasing.

- `either`: the signal crosses zero while either increasing or decreasing.
- `function-call`: the signal send a function call.

Note "Computing Operating Point Snapshots at Triggered Events" illustrates how to use this block.

See Also

`findop` | `linearize`

Topics

"Computing Operating Point Snapshots at Triggered Events"

Introduced before R2006a

Objects — Alphabetical List

BlockDiagnostic

Diagnostic information for individual block linearization

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains `BlockDiagnostic` objects. Each `BlockDiagnostic` object contains diagnostic information regarding the linearization of the corresponding Simulink block. You can troubleshoot the block linearization by examining the `BlockDiagnostic` object properties.

Creation

To access block diagnostic information in a `LinearizationAdvisor` object, use the `getBlockInfo` function. Using this function, you can obtain either a single `BlockDiagnostic` object or multiple `BlockDiagnostic` objects. For example, see:

- “Obtain Diagnostics for Potentially Problematic Blocks” on page 15-4
- “Obtain Diagnostics Using Block Names” on page 15-5

Properties

`IsOnPath` — Flag indicating whether the block is on the linearization path

`'Yes' | 'No'`

Flag indicating whether the block is on the linearization path, specified as one of the following:

- `'Yes'` — Block is on linearization path
- `'No'` — Block is not on linearization path

The linearization path connects the linearization inputs to the linearization outputs. To view the linearization path in the Simulink model, use the `highlight` function.

`ContributesToLinearization` — Flag indicating whether the block numerically influences the model linearization
`'Yes'` | `'No'`

Flag indicating whether the block numerically influences the model linearization, specified as one of the following:

- `'Yes'` — Block contributes to the linearization result
- `'No'` — Block does not contribute to the linearization result

If a block is not on the linearization path; that is, if `IsOnPath` is `'No'`, then `ContributesToLinearization` is `'No'`.

`DiagnosticMessages` — Diagnostic messages
cell array of character vectors

Diagnostic message regarding the block linearization, specified as a cell array of character vectors. These messages indicate possible issues that can affect the block linearization.

If `HasDiagnostics` is `'No'`, then `DiagnosticMessages` is an empty cell array.

`BlockPath` — Block path
character vector

Block path in Simulink model, specified as a character vector.

`HasDiagnostics` — Flag indicating whether the block has diagnostic messages
`'Yes'` | `'No'`

Flag indicating whether the block has diagnostic messages regarding its linearization, specified as one of the following:

- `'Yes'` — Block has diagnostic messages
- `'No'` — Block does not have diagnostic messages

If `HasDiagnostics` is `'Yes'`, then `DiagnosticMessages` is a cell array of character vectors that contains the messages.

`Linearization` — Block linearization
state-space model

Block linearization, specified as a state-space model.

`LinearizationMethod` — Linearization method
`'Exact'` | `'Perturbation'` | `'Block Substituted'` | `'Simscape Engine'` | `'Not Supported'`

Linearization method, specified as one of the following:

- `'Exact'` — Block linearized using its defined exact linearization
- `'Perturbation'` — Block linearized using numerical perturbation
- `'Block Substituted'` — Block linearized using a specified custom linearization
- `'Simscape Engine'` — Simscape network linearized using the exact linearization defined in the Simscape engine. A `LinearizationAdvisor` object does not provide diagnostic information on a component-level basis for Simscape networks. Instead, it groups diagnostic information for multiple Simscape components together.
- `'Not Supported'` — Block in its current configuration does not support linearization. For example, a Discrete Transfer Fcn block with an external reset does not support linearization.

  In this case, the block `Linearization` is zero. For more troubleshooting information, check the `DiagnosticMessages` property.

`OperatingPoint` — Operating point
`BlockOperatingPoint` object

Operating point at which the block is linearized, specified as a `BlockOperatingPoint` object.

Usage

You can troubleshoot the linearization of a Simulink model by examining the diagnostics for individual block linearizations. To do so, examine the properties of `BlockDiagnostic` objects returned from `getBlockInfo`. For more information, see “Troubleshoot Linearization Results at Command Line” on page 4-40.

Examples

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)
```

```
diags =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Obtain Diagnostics Using Block Names

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
```

```
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the saturation block.

```
satDiag = getBlockInfo(advisor,'scdpendulum/pendulum/Saturation')
```

```
satDiag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostic information for multiple blocks at once. Obtain diagnostics for the sin blocks in the model.

```
sinBlocks = {'scdpendulum/pendulum/Trigonometric Function';
             'scdpendulum/angle_wrap/Trigonometric Function1'};

sinDiag = getBlockInfo(advisor,sinBlocks)
```

```
sinDiag =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

See Also

Using Objects

`BlockOperatingPoint` | `LinearizationAdvisor`

Functions

`getBlockInfo` | `getBlockPaths` | `highlight`

Topics

“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

BlockOperatingPoint

Operating point at which block is linearized

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains `BlockDiagnostic` objects. Each `BlockDiagnostic` object contains diagnostic information regarding the linearization of the corresponding Simulink block. Each `BlockDiagnostic` object contains a `BlockOperatingPoint` with the input and state values for the operating point at which the block was linearized.

Creation

To obtain the operating point at which a block was linearized, use the `OperatingPoint` property of a `BlockDiagnostic` object. For example, see “Obtain Block Operating Point” on page 15-9.

Properties

`States` — Block state values

structure | structure array

State values at operating point, specified as a structure if the block has a single state, or a structure array if the block has multiple states. Each state structure has the following fields:

- `Name` — State name
- `x` — State value

`Inputs` — Block input values

structure | structure array

Input values at operating point, specified as a structure if the block has a single input, or a structure array if the block has multiple inputs. Each input structure has the following fields:

- `Port` — Input port number
- `u` — Input value

`BlockPath` — Block path

character vector

Block path in Simulink model, specified as a character vector.

Usage

When troubleshooting a block linearization, you can check the input and state values for the operating point at which the block was linearized using the `OperatingPoint` property of a `BlockDiagnostic` object.

Examples

Obtain Block Operating Point

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain a `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain block diagnostics for the second block in the list. This block is a second-order integrator.

```
diags = getBlockInfo(advisor,2);
```

Obtain the operating point at which this block was linearized.

```
blockOP = diags.OperatingPoint
```

```
blockOP =
  BlockOperatingPoint
```

The block has two states and one input.

See Also

Using Objects

`BlockDiagnostic`

Topics

“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

CompoundQuery

Complex query object for finding specific blocks in linearization results

Description

`CompoundQuery` query object for finding all the blocks in a `LinearizationAdvisor` object that have a specified number of inputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

To create a `CompoundQuery` object, combine other query objects using AND (`&`), OR (`|`), and NOT (`~`) logical operations. For example, see:

- "Find All SISO Blocks" on page 15-13
- "Create Complex Query Object" on page 15-12

Properties

`QueryType` — Query type

character vector

Query type, specified as a character vector. By default, `QueryType` is constructed using logical operators and the `QueryType` properties of the queries used to create the compound query. For example, suppose that you create a compound query for finding all SISO blocks:

```
qIn = linqueryHasInputs(1);
qOut = linqueryHasOutputs(1);
qSISO = qIn & qOut;
```

Then, `QueryType` is `'(Has 1 Inputs & Has 1 Outputs)'`.

You can modify `QueryType` for your application. For example:

```
qSISO.QueryType = 'SISO Blocks';
```

`Description` — Query description

`''` (default) | character vector

Query description, specified as `''` by default. You can add your own description to the query object using this property.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Create Complex Query Object

Create a `CompundQuery` object for finding any blocks that linearize to zero or any non-SISO blocks that are on the linearization path.

Create a query object for finding all non-SISO blocks.

```
qNotSISO = ~(linqueryHasOutputs(1) & linqueryHasInputs(1));
```

Create a query object for finding all blocks on the linearization path.

```
qOnPath = linqueryIsOnPath;
```

Create a query object for finding all blocks that linearize to zero.

```
qZero = linqueryIsZero;
```

To create a query for finding any blocks that linearize to zero or any non-SISO blocks that are on the linearization path, combine the other query objects.

```
query = (qNotSISO & qOnPath) | qZero

query =
  CompoundQuery with properties:

      QueryType: '((~((Has 1 Outputs & Has 1 Inputs)) & On Linearization Path) | Linear
    Description: ''
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)

advSISO =
  LinearizationAdvisor with properties:

              Model: 'scdspeed'
    OperatingPoint: [1x1 opcond.OperatingPoint]
  BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
         QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

LinearizationAdvisor

Diagnostic information for troubleshooting linearization results

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. You can troubleshoot your linearization results by reviewing this diagnostic information.

To access the diagnostic information, use the `getBlockInfo` function.

Creation

There are several ways to create a `LinearizationAdvisor` object when linearizing a Simulink model. When you linearize a model using:

- The `linearize` function, first create a `linearizeOptions` option set, setting the `StoreAdvisor` option to `true`. Then, linearize the model using `linearize`, returning the `info` argument.
- An `slLinearizer` interface, first create a `linearizeOptions` option set, setting the `StoreAdvisor` option to `true`. Then, create the `slLinearizer` interface. When you obtain a linear model from the interface using a linearization function, such as `getIOTransfer`, return the `info` argument.
- An `slTuner` interface, first create a `slTunerOptions` option set, setting the `StoreAdvisor` option to `true`. Then, create the `slTuner` interface. When you obtain a linear model from the interface using a linearization function, such as `getIOTransfer`, return the `info` argument.

You can then access the `LinearizationAdvisor` object using `info.Advisor`. If you linearize the model at multiple operating points or using parameter variation, `info.Advisor` is an array of `LinearizationAdvisor` objects.

Also, the `advise` and `find` functions return a `LinearizationAdvisor` object that contains diagnostic information for blocks that satisfy the relevant search criteria.

Properties

`Model` — Simulink model

character vector

Simulink model associated with the linearization diagnostic information, returned as a character vector.

`Model` is a read-only property.

`AnalysisPoints` — Linear analysis points

linearization I/O object | vector of linearization I/O objects

Linear analysis points, including inputs, outputs, and openings, returned as a linearization I/O object or a vector of linearization I/O objects.

`AnalysisPoints` corresponds to the:

- `io` input argument of the `linearize` command.
- Analysis points and loop openings of an `slLinearizer` or `slTuner` interface.

For more information on analysis points, see “Specify Portion of Model to Linearize” on page 2-13.

`AnalysisPoints` is a read-only property.

`OperatingPoint` — Operating point

operating point object

Operating point at which the model was linearized, specified as an operating point object.

`OperatingPoint` is a read-only property.

`Parameters` — Parameter samples

`[]` (default) | structure | structure array

Parameter samples for linearization, specified as one of the following:

- `[]` — Linearization result has no associated parameter values.
- Structure — Value for a single parameter, specified as a structure with the following fields:

  - `Name` — Parameter name
  - `Value` — Parameter value
- Structure array — Values for multiple parameters.

For more information on parameter variation, see “Specify Parameter Samples for Batch Linearization” on page 3-62.

`Parameters` is a read-only property.

`LinearizationOptions` — Linearization algorithm options

`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` object.

`LinearizationOptions` corresponds to the `options` input argument of `linearize`, `slLinearizer`, or `slTuner`.

`LinearizationOptions` is a read-only property.

`BlockDiagnostics` — Diagnostic information

`BlockDiagnostic` object | vector of `BlockDiagnostic` objects

Diagnostic information for each block that matches the search criteria used to create the `LinearizationAdvisor` object, specified as a `BlockDiagnostic` object or a vector of `BlockDiagnostic` objects.

You can access these block diagnostics using the `getBlockInfo` command. To obtain a list of the blocks, use the `getBlockPaths` command.

`BlockDiagnostics` is a read-only property.

`QueryType` — Query type

character vector

Query type used to obtain the linearization diagnostics, specified as one of the following:

- `'All Blocks'` when you initially create a `LinearizationAdvisor` object using a linearization function such as `linearize` or `getIOTransfer`.
- `'Linearization Advice'` when you create a `LinearizationAdvisor` object using the `advise` command.

- A character vector matching the `QueryType` property of the corresponding custom query object when you create a `LinearizationAdvisor` object using the `find` command.

`QueryType` is a read-only property.

`Description` — Query description

character vector

Description of the query used to obtain the linearization diagnostics, specified as one of the following:

- `'All Linearized Blocks'` when you initially create a `LinearizationAdvisor` object using a linearization function such as `linearize` or `getIOTransfer`.
- `'Blocks that are Potentially Problematic for Linearization'` when you create a `LinearizationAdvisor` object using the `advise` command.
- A character vector matching the `Description` property of the corresponding custom query object when you create a `LinearizationAdvisor` object using the `find` command.

`Description` is a read-only property.

Object Functions

advise	Find blocks that are potentially problematic for linearization
highlight	Highlight linearization path in Simulink model
find	Find blocks in linearization results that match specific criteria
getBlockInfo	Obtain diagnostic information for block linearizations
getBlockPaths	Obtain list of blocks in LinearizationAdvisor object

Examples

Create `LinearizationAdvisor` Using `linearize`

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Create a linearization option set, enabling the `StoreAdvisor` option.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Linearize the model using this option set, returning the `info` argument.

```
io = getlinio(mdl);
[linsys,~,info] = linearize(mdl,io,opt);
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x11 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Create `LinearizationAdvisor` Using `slLinearizer` Interface

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Create a linearization option set, enabling the `StoreAdvisor` option.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Define input and output analysis points, and create an `slLinearizer` interface using this option set.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
SL = slLinearizer(mdl,io,opt);
```

Find the transfer function from the input to the output, returning the `info` argument.

```
[linsys,info] = getIOTransfer(SL,'scdspeed/throttle (degrees)','scdspeed/rad//s to rpm'
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x27 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Create `LinearizationAdvisor` Using `slTuner` Interface

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Create a `slTunerOptions` option set, enabling the `StoreAdvisor` option.

```
opt = slTunerOptions('StoreAdvisor',true);
```

Define input and output analysis points, and create an `slTuner` interface using this option set.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
ST = slTuner(mdl,io,opt);
```

Typically, you would tune your control system using the `systune` function. Then, you can find the transfer function from the input to the output, returning the `info` argument.

```
[linsys,info] = getIOTransfer(ST,'scdspeed/throttle (degrees)','scdspeed/rad//s to rpm'
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:
```

```
                Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x27 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Find Potentially Problematic Blocks for Linearization

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find potentially problematic blocks for linearization.

```
result = advise(advisor)
```

```
result =
  LinearizationAdvisor with properties:

                Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearization Advice'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)

diags =
Linearization Diagnostics for the Blocks:

    IsOnPath
    ContributesToLinearization
    LinearizationMethod
    Linearization
    OperatingPoint
```

Alternative Functionality

App

You can interactively troubleshoot linearization results using the Linearization Advisor in the Linear Analysis Tool. For an example, see “Troubleshoot Linearization Results in Linear Analysis Tool” on page 4-21.

See Also

Using Objects

`BlockDiagnostic`

Functions

`advise` | `find` | `getCompSensitivity` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `linearize`

Topics

“Identify and Fix Common Linearization Issues” on page 4-8
“Troubleshoot Linearization Results at Command Line” on page 4-40
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52

Introduced in R2017b

linqueryAdvise

Query object for finding blocks that are potentially problematic for linearization

Description

`linqueryAdvise` creates a custom query object for finding the blocks in a linearization result that are potentially problematic for linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Using the `find` function with a `linqueryAdvise` object is equivalent to using the `advise` function.

Creation

Syntax

```
query = linqueryAdvise
```

Description

`query = linqueryAdvise` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are potentially problematic for linearization.

Properties

`QueryType` — Query type

`'Linearization Advice'` (default) | character vector

Query type, specified as `'Linearization Advice'`.

`Description` — Query description

`'Blocks that are Potentially Problematic for Linearization'` (default) | character vector

Query description, specified as `'Blocks that are Potentially Problematic for Linearization'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryAdvise` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are potentially problematic for linearization by using the `linqueryAdvise` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryAdvise` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Potentially Problematic Linearizations

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks that have potentially problematic linearizations.

```
qAdvise = linqueryAdvise;
advAdvise = find(advisor,qAdvise)
```

```
advAdvise =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearization Advice'
```

Algorithms

Creating a `linqueryAdvise` object is equivalent to creating the following custom query:

```
qPath     = linqueryIsOnPath;
qZero     = linqueryIsZero;
qBlkRep   = linqueryIsBlockSubstituted;
qDiags    = linqueryHasDiagnostics;

q = qPath & (qZero | qDiags | qBlkRep);

advisor_new = find(advisor,q);
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryAllBlocks

Query object for finding all linearized blocks

Description

`linqueryAllBlocks` creates a custom query object for finding all the linearized blocks listed in a `LinearizationAdvisor` object.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

When you use this query object with the `find` command, the `LinearizationAdvisor` object returned by `find` contains the same blocks as the input `LinearizationAdvisor` object. Therefore, it is not necessary to use `linqueryAllBlocks`. This command is a utility function used by the Linearization Advisor in the Linear Analysis Tool.

Creation

Syntax

```
query = linqueryAllBlocks
```

Description

`query = linqueryAllBlocks` creates a query object for finding all the linearized blocks listed in a `LinearizationAdvisor` object.

Properties

`QueryType` — Query type

`'All Blocks'` (default) | character vector

Query type, specified as `'All Blocks'`.

`Description` — Query description

`'All Linearized Blocks'` (default) | character vector

Query description, specified as `'All Linearized Blocks'`.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Linearized Blocks

Load the Simulink model.

```
mdl = 'scdpwm';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
[sys,op,info] = linearize(mdl,getlinio(mdl),opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks.

```
qAll = linqueryAllBlocks;
advAll = find(advisor,qAll)
```

```
advAll =
  LinearizationAdvisor with properties:

               Model: 'scdpwm'
```

```
    OperatingPoint: [1x1 opcond.OperatingPoint]
BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
       QueryType: 'All Blocks'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryContributesToLinearization

Query object for finding blocks that contribute to the model linearization result

Description

`linqueryContributesToLinearization` creates a custom query object for finding all the blocks that numerically contribute to the model linearization result.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryContributesToLinearization
```

Description

`query = linqueryContributesToLinearization` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that numerically contribute to the model linearization result.

Properties

`QueryType` — Query type

`'Contributes to Linearization'` (default) | character vector

Query type, specified as `'Contributes to Linearization'`.

`Description` — Query description

`'Blocks that Contribute to the Model Linearization'` (default) | character vector

Query description, specified as `'Blocks that Contribute to the Model Linearization'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryContributesToLinearization` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that numerically contribute to the model linearization result by using the `linqueryContributesToLinearization` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryContributesToLinearization` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks That Contribute to Linearization Result

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
```

```
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object and find all blocks that numerically contribute to the model linearization result.

```
qContribute = linqueryContributesToLinearization;
advContribute = find(advisor,qContribute)
```

```
advContribute =
  LinearizationAdvisor with properties:

              Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x22 linearize.advisor.BlockDiagnostic]
           QueryType: 'Contributes to Linearization'
```

To find blocks that do not contribute to the linearization result, use the same query object with a NOT (~) logical operator.

```
advNoContribute = find(advisor,~qContribute)
```

```
advNoContribute =
  LinearizationAdvisor with properties:

              Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x5 linearize.advisor.BlockDiagnostic]
           QueryType: '~(Contributes to Linearization)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find` | `highlight`

Topics

"Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52
"Troubleshoot Linearization Results at Command Line" on page 4-40

Introduced in R2017b

linqueryHasDiagnostics

Query object for finding blocks that have diagnostic messages regarding their linearization

Description

`linqueryHasDiagnostics` creates a custom query object for finding all the blocks in a linearization result that have diagnostic messages regarding their linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryHasDiagnostics
```

Description

`query = linqueryHasDiagnostics` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have diagnostic messages regarding their linearization.

Properties

`QueryType` — Query type

`'Has Diagnostics'` (default) | character vector

Query type, specified as `'Has Diagnostics'`.

`Description` — Query description

`'Blocks that have Linearization Diagnostic Messages'` (default) | character vector

Query description, specified as `'Blocks that have Linearization Diagnostic Messages'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasDiagnostics` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have diagnostic messages regarding their linearization by using the `linqueryHasDiagnostics` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasDiagnostics` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Linearization Diagnostic Messages

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with diagnostic messages regarding their linearization.

```
qDiag = linqueryHasDiagnostics;
advDiag = find(advisor,qDiag)
```

```
advDiag =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has Diagnostics'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

"Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52
"Troubleshoot Linearization Results at Command Line" on page 4-40

Introduced in R2017b

linqueryHasInputs

Query object for finding blocks with specified number of inputs

Description

`linqueryHasInputs` creates a custom query object for finding all the blocks in a linearization result that have a specified number of inputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryHasInputs(Nu)
```

Description

`query = linqueryHasInputs(Nu)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have `Nu` inputs.

Input Arguments

`Nu` — Number of block inputs

nonnegative integer

Number of block inputs, specified as a nonnegative integer.

Properties

`NumInputs` — Number of block inputs
nonnegative integer

Number of block inputs, specified as a nonnegative integer equal to `Nu`.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> Inputs'`, where `<N>` is equal to `NumInputs`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> Inputs'`, where `<N>` is equal to `NumInputs`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasInputs` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of inputs by using the `linqueryHasInputs` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasInputs` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two Inputs

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object and find all the linearized blocks with two inputs.

```
qIn = linqueryHasInputs(2);
advIn = find(advisor,qIn)
```

```
advIn =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x13 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 Inputs'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)

advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryHasOrder

Query object for finding blocks with specified number of states

Description

`linqueryHasOrder` creates a custom query object for finding all the blocks in a linearization result that have a specified number of states.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryHasStates(Nx)
```

Description

`query = linqueryHasStates(Nx)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have `Nx` states.

Input Arguments

`Nx` — Number of block states

nonnegative integer

Number of block states, specified as a nonnegative integer.

Properties

`NumStates` — Number of block states
nonnegative integer

Number of block states, specified as a nonnegative integer equal to `Nx`.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> States'`, where `<N>` is equal to `NumStates`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> States`, where `<N>` is equal to `NumStates`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasOrder` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of states by using the `linqueryHasOrder` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasOrder` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two States

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks with two states.

```
qOrder = linqueryHasOrder(2);
advOrder = find(advisor,qOrder)
```

```
advOrder =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 States'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

"Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52
"Troubleshoot Linearization Results at Command Line" on page 4-40

Introduced in R2017b

linqueryHasOutputs

Query object for finding blocks with specified number of outputs

Description

`linqueryHasOutputs` creates a custom query object for finding all the blocks in a linearization result that have a specified number of outputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryHasOutputs(Ny)
```

Description

`query = linqueryHasOutputs(Ny)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have `Ny` outputs.

Input Arguments

`Ny` — Number of block outputs

nonnegative integer

Number of block outputs, specified as a nonnegative integer.

Properties

`NumOutputs` — Number of block outputs
nonnegative integer

Number of block outputs, specified as a nonnegative integer equal to `Ny`.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> Outputs'`, where `<N>` is equal to `NumOutputs`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> Outputs'`, where `<N>` is equal to `NumOutputs`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasOutputs` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of outputs by using the `linqueryHasOutputs` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasOutputs` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two Outputs

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks with two outputs.

```
qOut = linqueryHasOutputs(2);
advOut = find(advisor,qOut)
```

```
advOut =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 Outputs'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

              Model: 'scdspeed'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
          QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryHasSampleTime

Query object for finding blocks with specified sample time

Description

`linqueryHasSampleTime` creates a custom query object for finding all the blocks in a linearization result that have a specified sample time.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryHasSampleTime(SampleTime)
```

Description

`query = linqueryHasSampleTime(SampleTime)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have a sample time equal to `SampleTime`.

Input Arguments

`SampleTime` — Block sample time
nonnegative scalar

Block sample time, specified as a nonnegative scalar. Specify `SampleTime` in the time units of the linearized model.

To find continuous-time blocks, specify `SampleTime` as `0`.

Properties

`Ts` — Sample

nonnegative scalar

Number of block outputs, specified as a nonnegative integer equal to `SampleTime`.

`QueryType` — Query type

character vector

Query type, specified as a character vector of the form `'Has <T> Sample Time'`, where `<T>` is equal to `Ts`.

`Description` — Query description

character vector

Query description, specified as a character vector of the form `'Blocks with <T> Sample Time'`, where `<T>` is equal to `Ts`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasSampleTime` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have a specified sample time by using the `linqueryHasSampleTime` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasSampleTime` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Specified Sample Time

Load the Simulink model.

```
mdl = 'scdmrate';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdmrate/Constant',1,'input');
io(2) = linio('scdmrate/sysTs2',1,'openoutput');
[linsys,linop,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object and find all the linearized blocks with a sample time of `0.1` seconds.

```
qTs = linqueryHasSampleTime(0.01);
advTs = find(advisor,qTs)
```

```
advTs =
  LinearizationAdvisor with properties:

               Model: 'scdmrate'
      OperatingPoint: [1×1 opcond.OperatingPoint]
    BlockDiagnostics: [1×1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 0.01 Sample Time'

Description: Blocks with 0.01 Sample Time

Check the 'BlockDiagnostic' property to examine the block diagnostics.
Use the advise or find command to further prune the block diagnostics.
```

Find All Continuous-Time Blocks

Load the Simulink model.

```
mdl = 'scdmrate';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all linearized blocks with continuous-time linearizations.

```
qCont = linqueryHasSampleTime(0);
advCont = find(advisor,qCont)
```

```
advCont =
  LinearizationAdvisor with properties:

               Model: 'scdmrate'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x5 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 0 Sample Time'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryHasZeroIOPair

Query object for finding blocks with at least one input/output pair that linearizes to zero

Description

`linqueryHasZeroIOPair` creates a custom query object for finding all the blocks in a linearization result that have at least one input/output pair that linearizes to zero. For a zero input/output pair, a change in the input value has no effect on the output value.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryHasZeroIOPair
```

Description

`query = linqueryHasZeroIOPair` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have at least one input/output path that linearizes to zero.

Properties

`QueryType` — Query type

`'Has Zero I/O Pair'` (default) | character vector

Query type, specified as `'Has Zero I/O Pair'`.

`Description` — Query description

`'Blocks with a Zero IO Pair'` (default) | character vector

Query description, specified as `'Blocks with a Zero IO Pair'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasZeroIOPair` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have at least one input/output path that linearizes to zero by using the `linqueryHasZeroIOPair` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasZeroIOPair` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Zero Input/Output Paths

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with at least one input/output path that linearizes to zero.

```
qZeroPair = linqueryHasZeroIOPair;
advZeroPair = find(advisor,qZeroPair)
```

```
advZeroPair =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x6 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has Zero I/O Pair'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsBlockSubstituted

Query object for finding blocks that have custom block linearizations specified

Description

`linqueryIsBlockSubstituted` creates a custom query object for finding all the blocks in a linearization result that have custom block linearizations specified.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryIsBlockSubstituted
```

Description

`query = linqueryIsBlockSubstituted` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have custom block linearization specified.

Properties

`QueryType` — Query type

`'Block Substituted'` (default) | character vector

Query type, specified as `'Block Substituted'`.

`Description` — Query description

`'Blocks Linearized with Block Substitution'` (default) | character vector

Query description, specified as `'Blocks Linearized with Block Substitution'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsBlockSubstituted` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have a custom linearization specified by using the `linqueryIsBlockSubstituted` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsBlockSubstituted` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Substitute Linearizations

Load the Simulink model.

```
mdl = 'scdpwmCustom';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
[sys,op,info] = linearize(mdl,getlinio(mdl),opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with substitute linearizations.

```
qSub = linqueryIsBlockSubstituted;
advSub = find(advisor,qSub)

advSub =
  LinearizationAdvisor with properties:

              Model: 'scdpwmCustom'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
          QueryType: 'Block Substituted'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsBlockType

Query object for finding blocks of the specified type

Description

`linqueryIsBlockType` creates a custom query object for finding all the blocks of a specified type in a linearization result.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueriIsBlockType(Type)
```

Description

`query = linqueriIsBlockType(Type)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are of type`Type`.

Input Arguments

`Type` — Block type
character vector | string

Block type, specified as a character vector or string. To specify a block type, use the corresponding `blocktype` parameter of the block. To get the `blocktype` parameter for the currently selected block in the Simulink model, at the MATLAB command line, type:

```
get_param(gcb,'blocktype')
```

Also, to find:

- MATLAB Function blocks, specify `Type` as `'matlab function'`.
- Stateflow charts, specify `Type` as `'chart'`.
- Simscape engine blocks, specify `Type` as `'simscape'`. A `LinearizationAdvisor` object does not provide diagnostic information on a component-level basis for Simscape networks. Instead, it groups diagnostic information for multiple Simscape components together.

Properties

`QueryType` — Query type

character vector

Query type, specified as a character vector of the form `'<type> Blocks'`, where `<type>` is equal the block type specified in `Type`.

`Description` — Query description

character vector

Query description, specified as a character vector of the form `'Blocks with <type> Block types'`, where `<type>` is equal to `Type`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsBlockType` query object, you can:

- Find all the blocks of a specified type in a `LinearizationAdvisor` object by using the `linqueryIsBlockType` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsBlockType` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Integrator Blocks in Linearization Result

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the integrator blocks.

```
qInteg = linqueryIsBlockType('Integrator');
advInteg = find(advisor,qInteg)
```

```
advInteg =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x2 linearize.advisor.BlockDiagnostic]
           QueryType: 'Integrator Blocks'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsExact

Query object for finding blocks linearized using their defined exact linearization

Description

`linqueryIsExact` creates a custom query object for finding all the blocks in a linearization result that are linearized using their defined exact linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryIsExact
```

Description

`query = linqueryIsExact` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are linearized using their defined exact linearization.

Properties

`QueryType` — Query type

`'Exact'` (default) | character vector

Query type, specified as `'Exact'`.

`Description` — Query description

`'Blocks that are Analytically Linearized'` (default) | character vector

Query description, specified as `'Blocks that are Analytically Linearized'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsExact` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are linearized using their defined exact linearization by using the `linqueryIsExact` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsExact` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks Linearized Using Exact Linearization

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks linearized using their defined exact linearization.

```
qExact = linqueryIsExact;
advExact = find(advisor,qExact)

advExact =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x21 linearize.advisor.BlockDiagnostic]
           QueryType: 'Exact'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsNumericallyPerturbed

Query object for finding blocks linearized using numerical perturbation

Description

`linqueryIsNumericallyPerturbed` creates a custom query object for finding all the blocks in a linearization result that are linearized using numerical perturbation.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see "Find Blocks in Linearization Results Matching Specific Criteria" on page 4-52.

Creation

Syntax

```
query = linqueryIsNumericallyPerturbed
```

Description

`query = linqueryIsNumericallyPerturbed` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are linearized using numerical perturbation.

Properties

`QueryType` — Query type

`'Perturbation'` (default) | character vector

Query type, specified as `'Perturbation'`.

`Description` — Query description

`'Blocks that are Numerically Perturbed'` (default) | character vector

Query description, specified as `'Blocks that are Numerically Perturbed'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsNumericallyPerturbed` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are linearized using numerical perturbation by using the `linqueryIsNumericallyPerturbed` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsNumericallyPerturbed` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Numerically Perturbed Blocks

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all numerically perturbed blocks.

```
qPert = linqueryIsNumericallyPerturbed;
advPert = find(advisor,qPert)

advPert =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x4 linearize.advisor.BlockDiagnostic]
           QueryType: 'Perturbation'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsOnPath

Query object for finding blocks that are on the linearization path

Description

`linqueryIsOnPath` creates a custom query object for finding all the blocks in a linearization result that are on the linearization path.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryIsOnPath
```

Description

`query = linqueryIsOnPath` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are on the linearization path.

Properties

`QueryType` — Query type

`'On Linearization Path'` (default) | character vector

Query type, specified as `'On Linearization Path'`.

`Description` — Query description

`'Blocks on the Linearization Path'` (default) | character vector

Query description, specified as `'Blocks on the Linearization Path'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsOnPath` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are on the linearization path by using the `linqueryIsOnPath` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsOnPath` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks On Linearization Path

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks on the linearization path.

```
qPath = linqueryIsOnPath;
advPath = find(advisor,qPath)

advPath =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x26 linearize.advisor.BlockDiagnostic]
           QueryType: 'On Linearization Path'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find` | `highlight`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

linqueryIsZero

Query object for finding blocks that linearize to zero

Description

`linqueryIsZero` creates a custom query object for finding all the blocks in a linearization result that linearize to zero.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the Linear Analysis Tool. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

Creation

Syntax

```
query = linqueryIsZero
```

Description

`query = linqueryIsZero` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that linearize to zero.

Properties

`QueryType` — Query type

`'Linearized to Zero'` (default) | character vector

Query type, specified as `'Linearized to Zero'`.

`Description` — Query description

`'Blocks Linearized to Zero'` (default) | character vector

Query description, specified as `'Blocks Linearized to Zero'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsZero` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that linearize to zero by using the `linqueryIsZero` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsZero` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks That Linearize to Zero

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks that linearize to zero.

```
qZero = linqueryIsZero;
advZero = find(advisor,qZero)

advZero =
  LinearizationAdvisor with properties:

              Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearized to Zero'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the Linear Analysis Tool. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52.

See Also

Using Objects

`CompoundQuery` | `LinearizationAdvisor`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-52
“Troubleshoot Linearization Results at Command Line” on page 4-40

Introduced in R2017b

Model Advisor Checks

Simulink Control Design Checks

Identify time-varying source blocks interfering with frequency response estimation

Identify all time-varying source blocks in the signal path of any output linearization point marked in the Simulink model.

Description

Frequency response estimation uses the steady-state response of a Simulink model to a specified input signal. Time-varying source blocks in the signal path prevent the response from reaching steady-state. In addition, when such blocks appear in the signal path, the resulting response is not purely a response to the specified input signal. Thus, time-varying source blocks can interfere with accurate frequency response estimation.

This check finds and reports all the time-varying source blocks which appear in the signal path of any output linearization output points currently marked on the Simulink model. The report:

- Includes blocks in subsystems and in referenced models that are in normal simulation mode
- Excludes any blocks specified as `BlocksToHoldConstant` in the `frestimateOptions` object you enter as the input parameter

For more information about the algorithm that identifies time-varying source blocks, see the `frest.findSources` reference page.

Available with Simulink Control Design.

Input Parameters

FRESTIMATE options object to compare results against

Provide the paths of any blocks to exclude from the check. Specify the block paths as an array of `Simulink.BlockPath` objects. This array is stored in the `BlocksToHoldConstant` field of an option set you create with `frestimateOptions`. See the `frestimateOptions` reference page for more information.

Results and Recommended Actions

Condition	Recommended Action

Source blocks exist whose output reaches linearization output points currently marked on the model.	Consider holding these source blocks constant during frequency response estimation.
	
	Use the `frest.findSources` command to identify time-varying source blocks at the command line. Then use the `BlocksToHoldConstant` option of `frestimateOptions` to pass these blocks to the `frestimate` command. For example,
	
	`% Get linearization I/Os from the model.`
	`mdl = 'scdengine';`
	`io = getlinio(mdl);`
	`% Find time-varying source blocks.`
	`blks = frest.findSources(mdl,io);`
	`% Create options set with blocks to hold constant.`
	`opts = frestimateOptions;`
	`opts.BlocksToHoldConstant = blks;`
	`% Run estimation with the options.`
	`in = frest.Sinestream;`
	`sysest = frestimate(mdl,io,in,opts);`
	
	For more information and examples, see the `frest.findSources` and `frestimateOptions` reference pages.

Tip

Sometimes, the model includes referenced models containing source blocks in the signal path of an output linearization point. In such cases, set the referenced models to normal simulation mode to ensure that this check locates them. Use the `set_param` command to set `SimulationMode` of any referenced models to `Normal` before running the check.

See Also

- "Estimate Frequency Response Using Linear Analysis Tool" on page 5-26
- "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 5-58
- `frest.findSources` reference page
- `frestimateOptions` reference page
- `frestimate` reference page