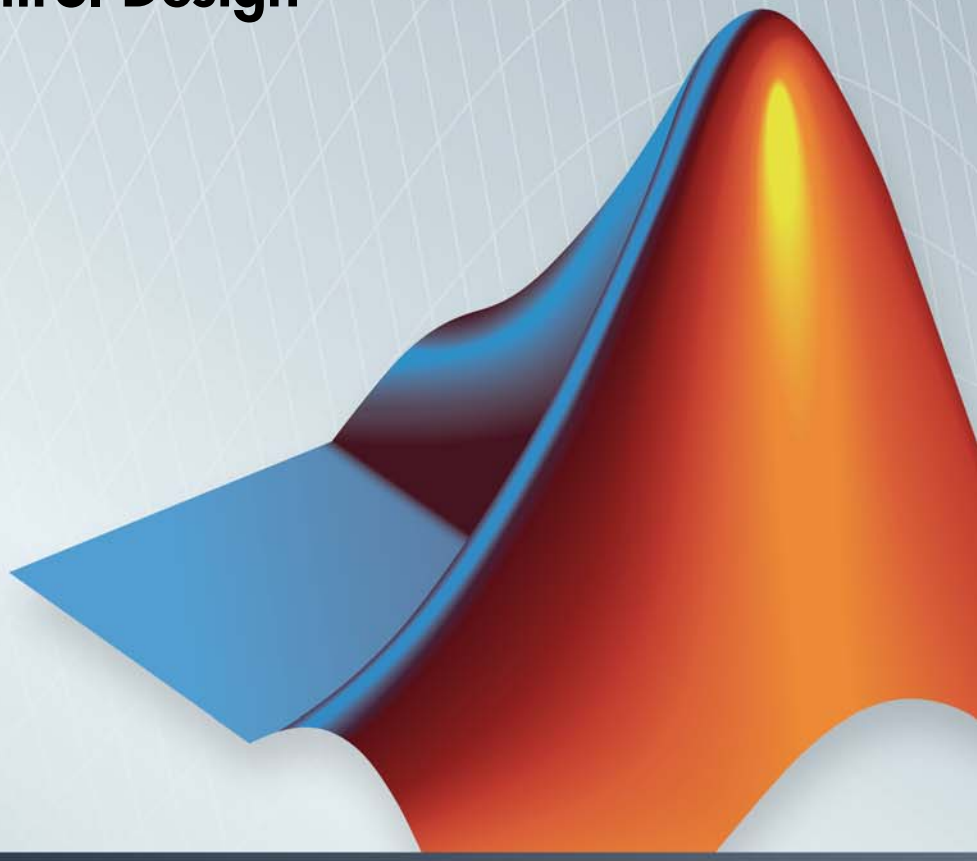
Simulink® Control Design™

User's Guide

R2013**b**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Simulink® Control Design™ User's Guide

© COPYRIGHT 2004–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)

Contents

1 Steady-State Operating Points

Steady-State Operating Point (Trimming) 1-2
What Is a Steady-State Operating Point? 1-2
What Is an Operating Point in Simulink Control Design? 1-3
Simulink Model States Included in Operating Point Object 1-4
Advantages of Using Simulink Control Design vs. Simulink Operating Point Search 1-5

View and Modify Operating Points 1-7
View Model Initial Condition in Linear Analysis Tool 1-7
Modify Operating Point in Linear Analysis Tool 1-8
View and Modify Operating Point Object (MATLAB Code) 1-10

Choosing Between Simulation Snapshot and Operating Point from Specifications 1-12

Steady-State Operating Points (Trimming) from Specifications 1-14
Steady-State Operating Point Search (Trimming) 1-14
Which States in the Model Must Be at Steady State? 1-15
Steady-State Operating Points from State Specifications . . 1-16
Steady-State Operating Point to Meet Output Specification 1-21
Initialize Steady-State Operating Point Search Using Simulation Snapshot 1-24
Compute Steady-State Operating Points for SimMechanics Models 1-29
Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code 1-31
Change Operating Point Search Optimization Settings . . . 1-34

Import and Export Specifications For Operating Point Search 1-36

Batch Compute Steady-State Operating Points 1-38

Steady-State Operating Points from Simulation 1-40
Simulation Snapshot Operating Points 1-40
Compute Operating Points at Simulation Snapshots 1-40

Simulate Simulink Model at Specific Operating Point 1-43

Handling Blocks with Internal State Representation . . 1-45
Operating Point Object Excludes Blocks with Internal States 1-45
Identifying Blocks with Internal States in Your Model . . . 1-46
Configuring Blocks with Internal States for Steady-State Operating Point Search 1-46

Synchronize Simulink Model Changes with Operating Point Specifications 1-48
Synchronize Simulink Model Changes with Linear Analysis Tool 1-48
Synchronize Simulink Model Changes with Existing Operating Point Specification Object 1-51

2 Linearization

Linearizing Nonlinear Models 2-3
What Is Linearization? 2-3
Applications of Linearization 2-5
Linearization in Simulink Control Design 2-6
Choosing Linearization Tools 2-7
Model Requirements for Exact Linearization 2-10
Operating Point Impact on Linearization 2-11

Specify Model Portion to Linearize 2-13

Specifying Subsystem, Loop, or Block to Linearize 2-13
Opening Feedback Loops 2-14
Ways to Specify Portion of Model to Linearize 2-16
Specify Portion of Model to Linearize in Simulink Model .. 2-17
Specify Portion of Model to Linearize in Linear Analysis Tool 2-20
Edit Portion of Model to Linearize in Linear Analysis Tool 2-25
Select Bus Elements as Linear Analysis Points 2-27

Plant Linearization 2-33

Open-Loop Response of Control System for Stability Margin Analysis 2-37
What Is Open-Loop Response? 2-37
Compute Open-Loop Response 2-38

Linearize at Model Operating Point 2-43
Linearize Simulink Model 2-43
Visualize Bode Response of Simulink Model During Simulation 2-47

Linearize at Trimmed Operating Point 2-57

Linearize at Simulation Snapshots and Triggered Events 2-62
Linearize at Simulation Snapshot 2-62
Linearize at Triggered Simulation Events 2-65
Visualize Linear System at Multiple Simulation Snapshots 2-69
Visualize Linear System of a Continuous-Time Model Discretized During Simulation 2-75
Visualize Linear System at Trigger-Based Simulation Events 2-80

Ordering States in Linearized Model 2-81
Control State Order of Linearized Model using Linear Analysis Tool 2-81
Control State Order of Linearized Model using MATLAB Code 2-84

Time-Domain Validation of Linearization **2-86**
Validate Linearization in Time Domain **2-86**
Choosing Time-Domain Validation Input Signal **2-88**

Frequency-Domain Validation of Linearization **2-90**
Validate Linearization in Frequency Domain using Linear Analysis Tool **2-90**
Choosing Frequency-Domain Validation Input Signal **2-94**

Visualize Models **2-96**
Customize Characteristics of Plot in Linear Analysis Tool **2-96**
Print Plot to MATLAB Figure in Linear Analysis Tool . . . **2-96**
Generate Additional Response Plots of Linearized System **2-97**
Add Linear System to Existing Response Plot **2-98**

Generate MATLAB Code for Linear Analysis Tool Session **2-99**

Troubleshooting Linearization **2-101**
Linearization Troubleshooting Overview **2-101**
Check Operating Point **2-109**
Check Linearization I/O Points Placement **2-109**
Check Loop Opening Placement **2-110**
Check Phase of Frequency Response for Models with Time Delays **2-110**
Check Individual Block Linearization Values **2-110**
Check Large Models **2-113**
Check Multirate Models **2-114**

Controlling Block Linearization **2-117**
When You Need to Specify Linearization for Individual Blocks **2-117**
Specify Linear System for Block Linearization Using MATLAB Expression **2-118**
Specify D-Matrix System for Block Linearization Using Function **2-119**
Augment the Linearization of a Block **2-123**
Models with Time Delays **2-127**
Perturbation Level of Blocks Perturbed During Linearization **2-129**

Linearizing Blocks with Nondouble Precision Data Type Signals 2-130
Event-Based Subsystems (Externally Scheduled Subsystems) 2-132

Models with Pulse Width Modulation (PWM) Signals . . 2-140

Specifying Linearization for Model Components Using System Identification 2-142

Speeding Up Linearization of Complex Models 2-150
Factors That Impact Linearization Performance 2-150
Blocks with Complex Initialization Functions 2-150
Disabling the Linearization Inspector in the Linear Analysis Tool 2-150
Batch Linearization of Large Simulink Models 2-151

Exact Linearization Algorithm 2-152
Continuous-Time Models 2-152
Multirate Models 2-153
Perturbation of Individual Blocks 2-154
User-Defined Blocks 2-156
Look Up Tables 2-157

How the Software Treats Loop Openings 2-158

What Is Batch Linearization? 2-160

Choosing Batch Linearization Tools 2-162

Batch Linearization Efficiency When You Vary Parameter Values 2-164

Batch Linearize Model for Parameter Value Variations Using linearize 2-165

Batch Linearize Model at Multiple Operating Points Using linearize 2-167

Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer **2-169**

Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer **2-177**

Use Response Plots to Analyze Batch Linearization Results **2-183**

Specify Parameter Samples **2-190**
Single Parameter **2-190**
Multiple Parameters **2-191**

Validate Batch Linearization Results **2-196**

3 Frequency Response Estimation

Frequency Response Model Applications **3-3**

What Is a Frequency Response Model? **3-4**

Model Requirements **3-6**

Estimation Requires Input and Output Signals **3-7**

Creating Input Signals for Estimation **3-9**
Supported Input Signals **3-9**
Creating Sinestream Input Signals **3-9**
Creating Chirp Input Signals **3-19**

Modifying Input Signals for Estimation **3-24**
Modifying Sinestream Input Signal Using Linear Analysis Tool **3-24**
Modifying Sinestream Input Signal (MATLAB Code) **3-26**

Estimate Frequency Response Using Linear Analysis Tool **3-28**

Estimate Frequency Response with Linearization-Based Input Using Linear Analysis Tool **3-32**

Estimate Frequency Response (MATLAB Code) **3-37**

Analyzing Estimated Frequency Response **3-41**
View Simulation Results **3-41**
Interpret Frequency Response Estimation Results **3-44**
Analyze Simulated Output and FFT at Specific Frequencies **3-46**
Annotate Frequency Response Estimation Plots **3-48**
Displaying Estimation Results for Multiple-Input Multiple-Output (MIMO) Systems **3-49**

Troubleshooting Frequency Response Estimation **3-50**
When to Troubleshoot **3-50**
Time Response Not at Steady State **3-50**
FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency **3-54**
Time Response Grows Without Bound **3-56**
Time Response Is Discontinuous or Zero **3-58**
Time Response Is Noisy **3-60**

Effects of Time-Varying Source Blocks on Frequency Response Estimation **3-63**
Setting Time-Varying Sources to Constant for Estimation Using Linear Analysis Tool **3-63**
Setting Time-Varying Sources to Constant for Estimation (MATLAB Code) **3-70**

Effects of Noise on Frequency Response Estimation . . **3-73**

Estimating Frequency Response Models with Noise Using Signal Processing Toolbox **3-75**

Estimating Frequency Response Models with Noise Using System Identification Toolbox **3-77**

Generate MATLAB Code for Repeated or Batch Frequency Response Estimation **3-79**

Managing Estimation Speed and Memory **3-81**
Ways to Speed up Frequency Response Estimation **3-81**
Speeding Up Estimation Using Parallel Computing **3-83**
Managing Memory During Frequency Response Estimation **3-87**

4 Designing Compensators

Choosing a Compensator Design Approach **4-2**

Introduction to Automatic PID Tuning **4-3**

What Plant Does the PID Tuner See? **4-5**

PID Tuning Algorithm **4-6**

Design Controllers with the PID Tuner **4-7**
Prerequisites for PID Tuning **4-7**
Opening the Tuner **4-8**
Analyzing the Design in the PID Tuner **4-10**
Refining the Design **4-12**
Verifying the PID Design in Your Simulink Model **4-13**
Tuning at a Different Operating Point **4-14**

Tune PID Controller to Balance Tracking and Disturbance Rejection Performance **4-18**

Design Two-Degree-of-Freedom PID Controllers **4-28**
About Two-Degree-of-Freedom PID Controllers **4-28**
Tuning Two-Degree-of-Freedom PID Controllers **4-28**

Tune PID Controller Within Model Reference **4-30**

Troubleshooting Automatic PID Tuning **4-33**
Plant Model Linearizes to Zero **4-33**
Cannot Find a Good Design in the PID Tuner **4-34**
Simulated Response Does Not Match the PID Tuner Response **4-35**
Cannot Find an Acceptable PID Design in the Simulated Model **4-36**
Controller Performance Deteriorates When Switching Time Domains **4-37**
When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain **4-38**

Designing a Simulink PID Controller (2DOF) Block for a Reactor **4-39**

Designing PID Controller in Simulink with Estimated Frequency Response **4-52**

Designing a Family of PID Controllers for Multiple Operating Points **4-60**

Design and Analysis of Control Systems **4-68**
Compensator Design Process Overview **4-68**
Beginning a Compensator Design Task **4-68**
Selecting Blocks to Tune **4-70**
Selecting Closed-Loop Responses to Design **4-73**
Selecting an Operating Point **4-75**
Creating a SISO Design Task **4-78**
Completing the Design **4-89**

5 Model Verification

Monitoring Linear System Characteristics in Simulink Models **5-2**

Defining a Linear System for Model Verification Blocks **5-4**

Verifiable Linear System Characteristics **5-5**

Model Verification at Default Simulation Snapshot Time **5-6**

Model Verification at Multiple Simulation Snapshots **5-15**

Model Verification Using Simulink Control Design and Simulink Verification Blocks **5-25**

6 Class Reference

7 Alphabetical List

8 Blocks — Alphabetical List

9 Model Advisor Checks

Simulink Control Design Checks **9-2**

Identify time-varying source blocks interfering with frequency response estimation **9-3**

Index

1

Steady-State Operating Points

- "Steady-State Operating Point (Trimming)" on page 1-2
- "View and Modify Operating Points" on page 1-7
- "Choosing Between Simulation Snapshot and Operating Point from Specifications" on page 1-12
- "Steady-State Operating Points (Trimming) from Specifications" on page 1-14
- "Import and Export Specifications For Operating Point Search" on page 1-36
- "Batch Compute Steady-State Operating Points" on page 1-38
- "Steady-State Operating Points from Simulation" on page 1-40
- "Simulate Simulink Model at Specific Operating Point" on page 1-43
- "Handling Blocks with Internal State Representation" on page 1-45
- "Synchronize Simulink Model Changes with Operating Point Specifications" on page 1-48

Steady-State Operating Point (Trimming)

In this section...

"What Is a Steady-State Operating Point?" on page 1-2

"What Is an Operating Point in Simulink® Control Design™?" on page 1-3

"Simulink Model States Included in Operating Point Object" on page 1-4

"Advantages of Using Simulink® Control Design™ vs. Simulink Operating Point Search" on page 1-5

What Is a Steady-State Operating Point?

An *operating point* of a dynamic system defines the overall *state* of this system 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.

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

A model might have several steady-state operating points. For example, a hanging pendulum has two steady-state operating points. A *stable steady-state operating point* occurs when a pendulum hangs straight down. That is, the pendulum position does not change with time. When the pendulum position deviates slightly, the pendulum always returns to equilibrium; 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 a nonlinear system, your initial guesses for the states and input levels must be in the neighborhood of the desired operating point to ensure convergence.

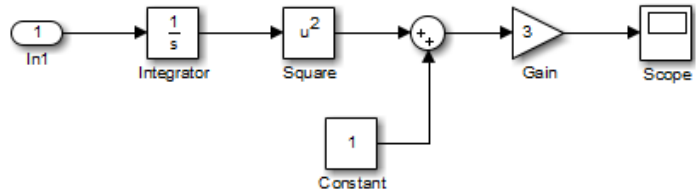
When linearizing a model with multiple steady-state operating points, it is important to have the right operating point. For example, linearizing a pendulum model around the stable steady-state operating point produces a stable linear model, whereas linearizing around the unstable steady-state operating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

What Is an Operating Point in Simulink Control Design?

The *operating point* of a model consists of the model initial states and root-level input signals.

For example, this Simulink® model has an operating point that consists of two variables:

- Root input level set to 1
- Integrator block state set to 5

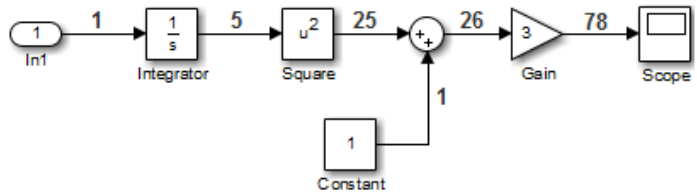


The next table summarizes the operating point values of this Simulink model.

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

Block	Block Input	Block Operation	Block Output
Sum	25 from Square block, 1 from Constant block	sums	26
Gain	26	multiplies by 3	78

The next 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.

Examples and How To

- “Steady-State Operating Points from State Specifications” on page 1-16
- “Steady-State Operating Point to Meet Output Specification” on page 1-21

More About

“Simulink Model States Included in Operating Point Object” on page 1-4

Simulink Model States Included in Operating Point Object

The operating point object in Simulink Control Design™ includes the tunable states in your Simulink model.

The operating point object excludes states of blocks that have internal representation, such as Backlash, Memory, and Stateflow blocks.

Block Type	Block Example	Included in Operating Point?
Blocks with double-precision real-valued states	Integrator, State Space, Transfer Function	Yes
Root-level inport blocks with double-precision real-valued inputs	Inport	Yes
Blocks with internal state representation that impact block output	Backlash, Memory, Stateflow	No

More About

- “Handling Blocks with Internal State Representation” on page 1-45
- “Steady-State Operating Point (Trimming)” on page 1-2

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

Simulink provides `trim` for steady-state operating point search. How is `trim` different from `findop` in Simulink Control Design for performing an optimization-based operating point search?

Simulink Control Design operating point search provides these advantages to using `trim`:

	Simulink Control Design Operating Point Search	Simulink Operating Point Search
Graphical-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
SimMechanics™ integration	Yes	No

View and Modify Operating Points

In this section...

"View Model Initial Condition in Linear Analysis Tool" on page 1-7

"Modify Operating Point in Linear Analysis Tool" on page 1-8

"View and Modify Operating Point Object (MATLAB Code)" on page 1-10

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.

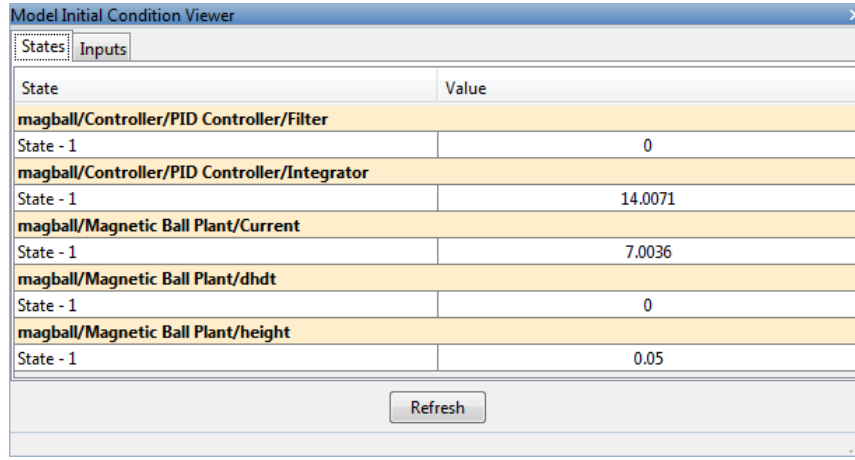
```
sys = 'magball';
open_system(sys)
```

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

The Linear Analysis Tool for the model opens.

3 Click in the **Exact Linearization** tab.

This action opens the Model Initial Condition Viewer, 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 Simulink model.

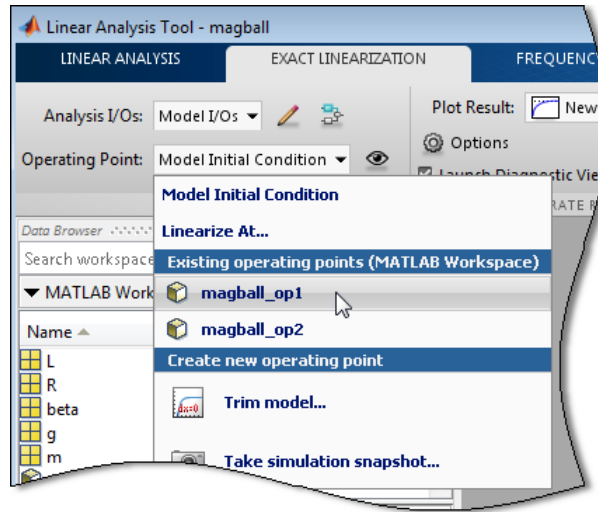
   ```
   sys = 'magball';
   open_system(sys)
   ```

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

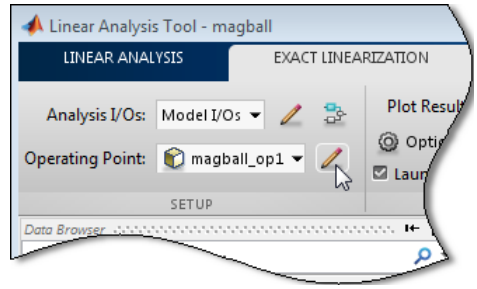
2. In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

   The Linear Analysis Tool for the model opens.

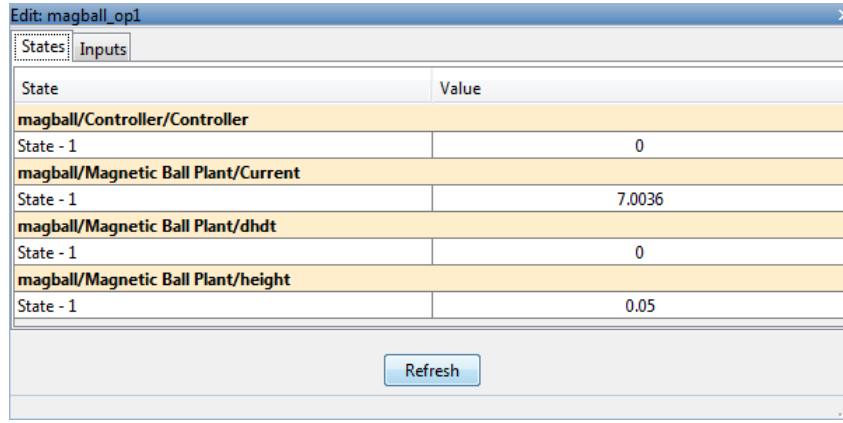
3. Choose `magball_op1` from the **Operating Point** list.



4 Click adjacent to the **Operating Point** list.



The `magball_op1` editor opens. Use this dialog box to view and edit this operating point.



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

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 the Simulink model using an operating point object.

1. Create operating point object from Simulink model.

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

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

2. Set the value of the first state.

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

3. View the operating point object state values.

```
op.States
```

```
(1.) watertank/PID Controller/Integrator
      x: 1.26
(2.) watertank/Water-Tank System/H
      x: 1
```

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

Choosing Between Simulation Snapshot and Operating Point from Specifications

You can find steady-state operating points (or trim conditions) from specifications or at specific simulation times (or simulation snapshots).

Choosing which approach to use for computing your operating point depends on what you know about the operating point.

Use *optimization-based steady-state operating point search* when you know some of the operating point states and model input or output signal levels. Successful operating point search finds an operating point very close to a true steady-state solution.

Optimization-based search produces poor results when you specify:

- 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.

  This is equivalent to *overconstraining* the optimization search.

Use the *simulation-based approach* when the simulation time is sufficiently short for the model to reach steady state. The algorithms extracts operating point values when the simulation reaches steady state. You must also specify the initial conditions that drive the model to steady state.

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

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

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.

Examples and How To

- "Steady-State Operating Point to Meet Output Specification" on page 1-21
- "Compute Operating Points at Simulation Snapshots" on page 1-40

More About

"Steady-State Operating Point (Trimming)" on page 1-2

Steady-State Operating Points (Trimming) from Specifications

In this section...

"Steady-State Operating Point Search (Trimming)" on page 1-14

"Which States in the Model Must Be at Steady State?" on page 1-15

"Steady-State Operating Points from State Specifications" on page 1-16

"Steady-State Operating Point to Meet Output Specification" on page 1-21

"Initialize Steady-State Operating Point Search Using Simulation Snapshot" on page 1-24

"Compute Steady-State Operating Points for SimMechanics Models" on page 1-29

"Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-31

"Change Operating Point Search Optimization Settings" on page 1-34

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 model input levels.

Optimization-based operating point computation requires you to specify initial guesses and constraints on the key operating point states, input levels, and model output signals.

You can usually improve your optimization results using simulation to initialize the optimization. For example, you can extract the initial values of the operating point at a simulation time when the model reaches the neighborhood of steady state.

Optimization-based operating point search lets you specify and constrain 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. You can overconstrain the optimization by specifying incompatible constraints or initial guesses that are far away from the desired solution.

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

Examples and How To

"Change Operating Point Search Optimization Settings" on page 1-34

More About

"Which States in the Model Must Be at Steady State?" on page 1-15

Which States in the Model Must Be at Steady State?

When configuring a steady-state operating point search, you do not always need to specify all states 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 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 steady state of the 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 require that all states should be at equilibrium.

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

Steady-State Operating Points from State Specifications

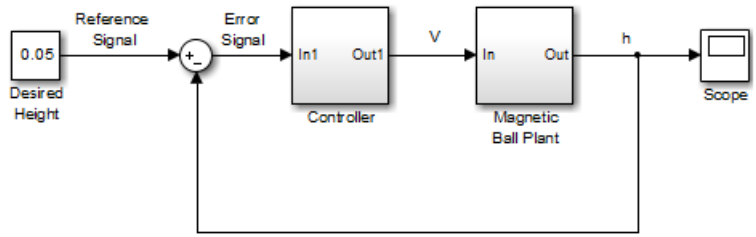
This example shows how to compute a steady-state operating point, or equilibrium operating point, by specifying known (fixed) equilibrium states and minimum state values.

Code Alternative

Use `findop` to find operating point from specifications. For examples and additional information, see the `findop` reference page.

1 Open Simulink model.

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



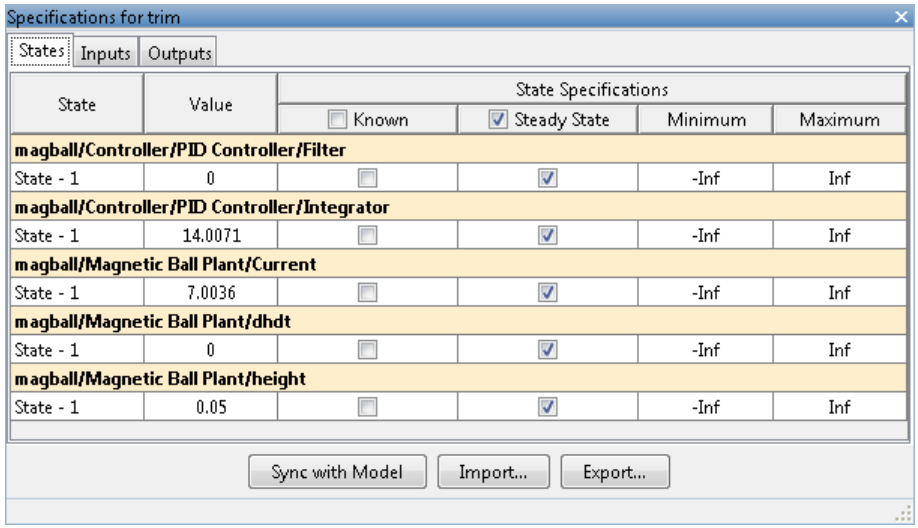
2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

The Linear Analysis Tool for the model opens.

3 In the **Linear Analysis** tab, click **Trim Model**. Then click **Specifications**.

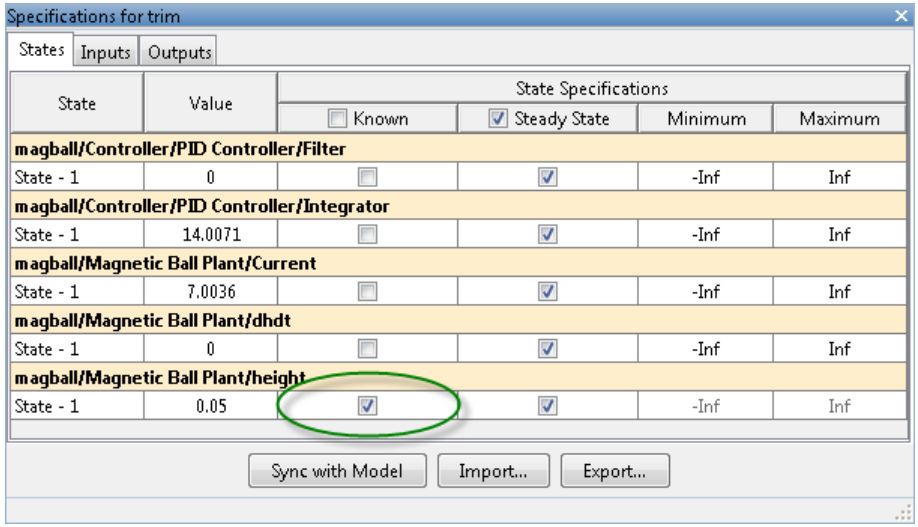
The Specifications for trim dialog box opens.

By default, the software specifies all model states to be at equilibrium (as shown by the check marks in the **Steady State** column). The **Inputs** and **Outputs** tabs are empty because this model does not have root-level input and output ports.

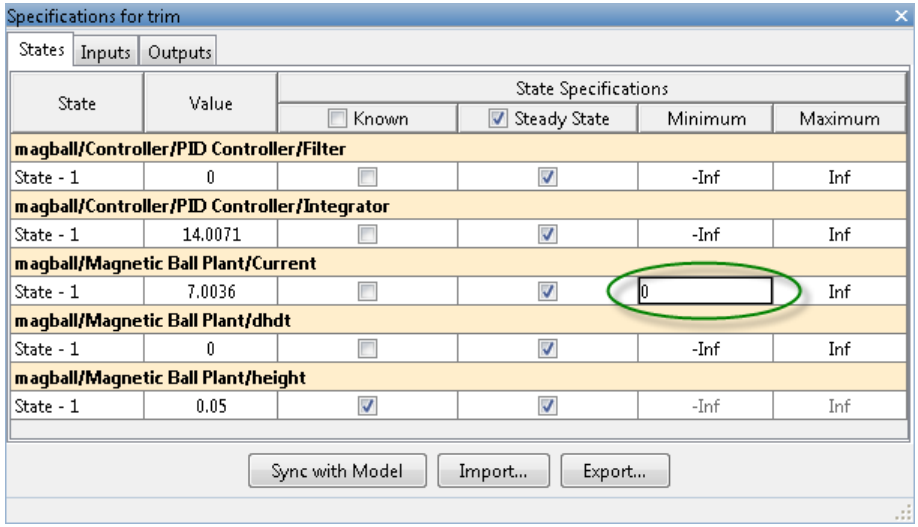


4 In the **States** tab, select **Known** for the **height** state.

The height of the ball matches the reference signal height (specified in the `Desired Height` block as `0.05`). This height value should remain fixed during the optimization.



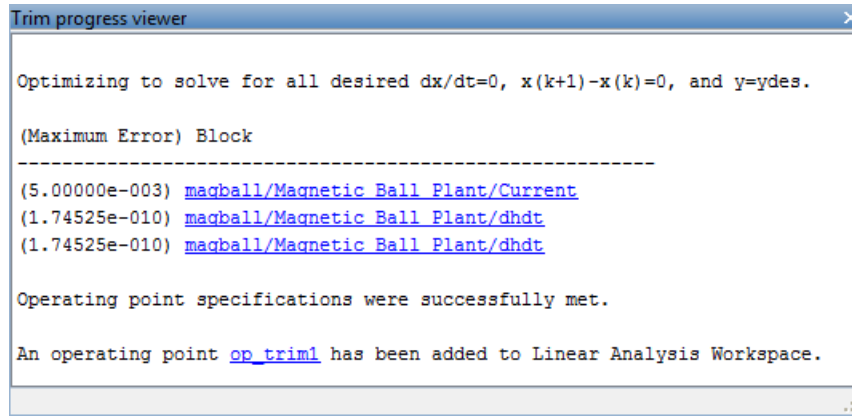
5 Enter 0 for the minimum bound of the **Current** state.



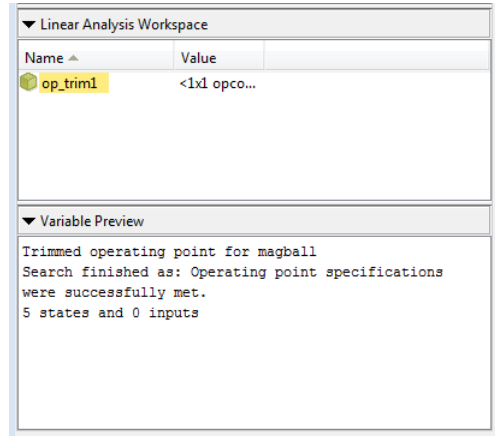
6 Click to compute the operating point.

This action uses numerical optimization to find the operating point that meets your specifications.

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.



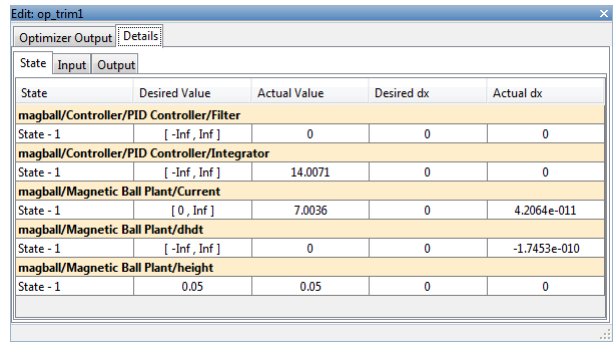
A new variable, `op_trim1`, appears in the **Linear Analysis Workspace**.



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

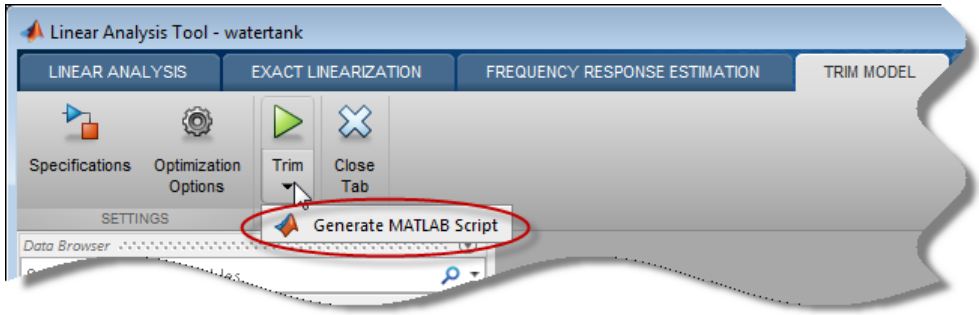
The **Actual dx** values are about 0, which indicates that the operating point meets the steady state specification.

The **Actual Value** of the states falls within the **Desired Value** bounds.



State	Desired Value	Actual Value	Desired dx	Actual dx
magball/Controller/PID Controller/Filter				
State - 1	[-Inf , Inf]	0	0	0
magball/Controller/PID Controller/Integrator				
State - 1	[-Inf , Inf]	14.0071	0	0
magball/Magnetic Ball Plant/Current				
State - 1	[0 , Inf]	7.0036	0	4.2064e-011
magball/Magnetic Ball Plant/dhdt				
State - 1	[-Inf , Inf]	0	0	-1.7453e-010
magball/Magnetic Ball Plant/height				
State - 1	0.05	0.05	0	0

8 (Optional) To automatically generate a MATLAB script, click **Trim** and select **Generate MATLAB Script**.



The generated script contains commands for computing the operating point for this example.

Related Examples

- "Steady-State Operating Point to Meet Output Specification" on page 1-21
- "Change Operating Point Search Optimization Settings" on page 1-34
- "Initialize Steady-State Operating Point Search Using Simulation Snapshot" on page 1-24

- "Compute Steady-State Operating Points for SimMechanics Models" on page 1-29
- "Simulate Simulink Model at Specific Operating Point" on page 1-43
- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-31

More About

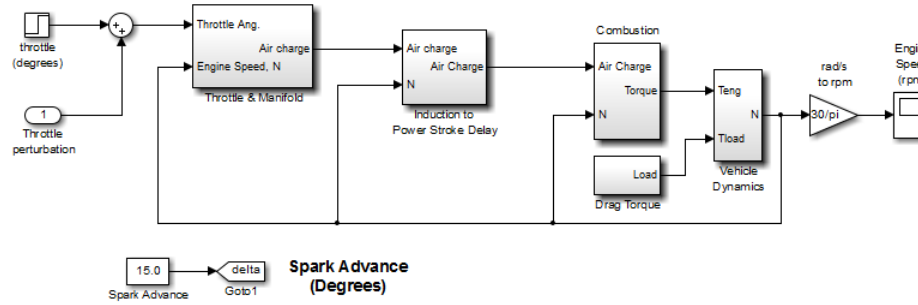
- "Steady-State Operating Point (Trimming)" on page 1-2
- "Choosing Between Simulation Snapshot and Operating Point from Specifications" on page 1-12

Steady-State Operating Point to Meet Output Specification

This example shows how to specify an output constraint of an engine speed for computing the engine steady-state operating point.

1 Open Simulink model.

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



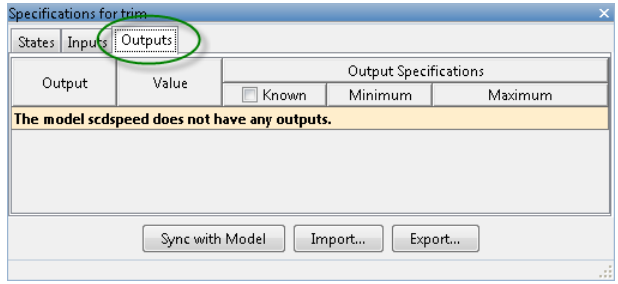
2 In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

The Linear Analysis Tool for the model opens.

3 In the **Linear Analysis** tab, click **Trim Model**. Then click **Specifications**.

The Specifications for trim dialog box appears.

4 Examine the linearization outputs for `scdspeed` in the **Outputs** tab.

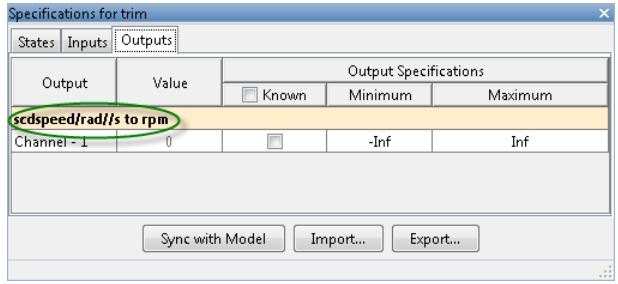


Currently there are no outputs specified for `scdspeed`.

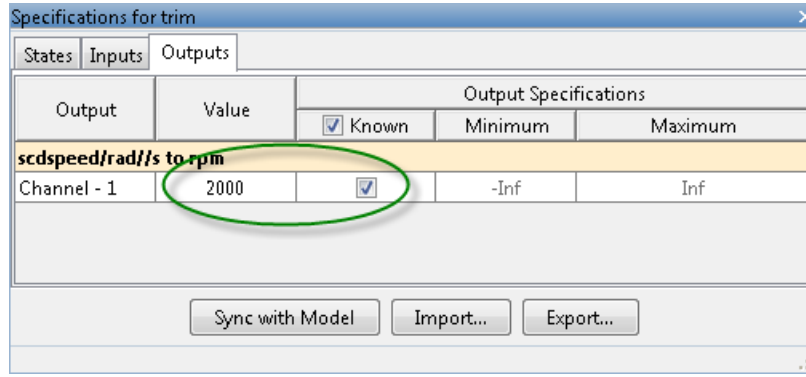
5 In the Simulink Editor, right-click the output signal from the `rad/s to rpm` block. Select **Linear Analysis Points > Trim Output Constraint**.

This action adds the output signal constraint marker `T` to the model.

The output signal from the `rad/s to rpm` block now appears under the **Outputs** tab.



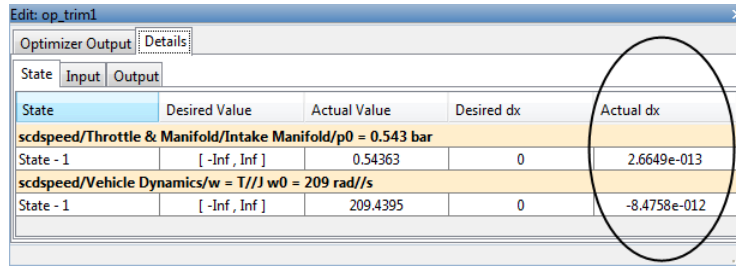
6 Select **Known** and enter `2000` RPM for the engine speed as the output signal value. Press **Enter**.



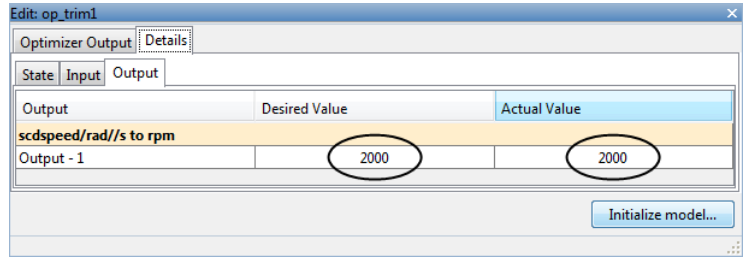
7 Click to find a new steady-state operating point that meets the specified output signal constraint.

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

In the **States** tab, the **Actual dx** values are either zero or about zero. This result indicates that the operating point meets the steady state specification.



In the **Outputs** tab, the **Actual Value** and the **Desired Value** are both 2000.



Related Examples

- "Steady-State Operating Points from State Specifications" on page 1-16
- "Change Operating Point Search Optimization Settings" on page 1-34
- "Initialize Steady-State Operating Point Search Using Simulation Snapshot" on page 1-24
- "Compute Steady-State Operating Points for SimMechanics Models" on page 1-29
- "Simulate Simulink Model at Specific Operating Point" on page 1-43
- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-31

More About

- "Steady-State Operating Point (Trimming)" on page 1-2
- "Choosing Between Simulation Snapshot and Operating Point from Specifications" on page 1-12

Initialize Steady-State Operating Point Search Using Simulation Snapshot

- "Initialize Operating Point Search Using Linear Analysis Tool" on page 1-25
- "Initialize Operating Point Search (MATLAB Code)" on page 1-27

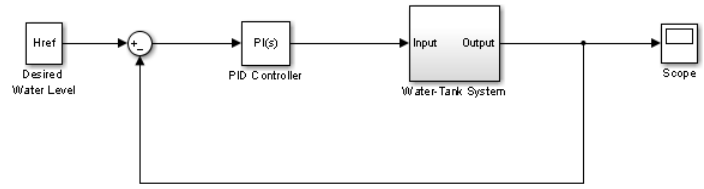
Initialize Operating Point Search Using Linear Analysis Tool

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 the state values to be used as the initial condition for numerical optimization.

1 Open Simulink model.

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



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

The Linear Analysis Tool for the model opens.

3 In the **Linear Analysis** tab, click **Operating Point Snapshot**

The **Operating Point Snapshots** tab opens.

4 Enter 10 in the **Simulation Snapshot Times** field to extract the operating point at this simulation time. Press **Enter**.

Click to take a snapshot of the system at the specified time.

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

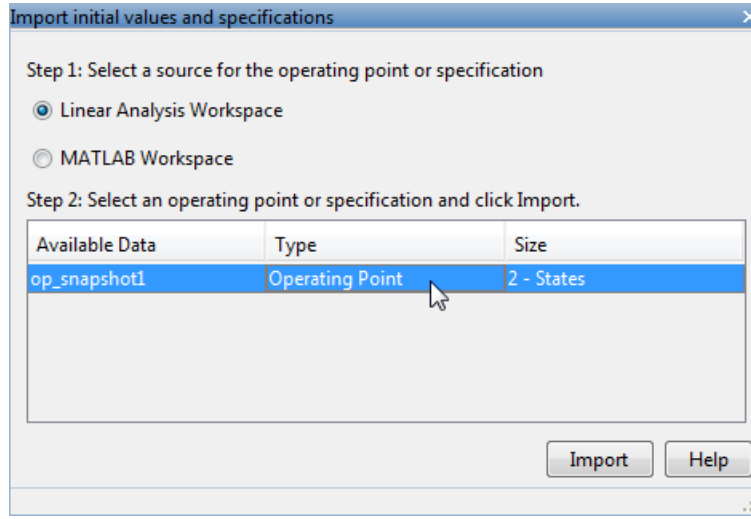
5 In the **Linear Analysis** tab, click **Trim Model**. Then click **Specifications**.

The Specifications for trim dialog box appears.

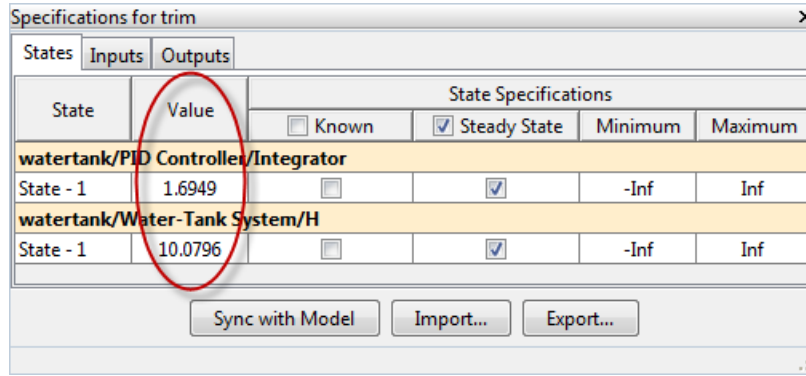
6 Click **Import**.

The Import initial values and specifications dialog opens.

7 Select `op_snapshot1` and click **Import** to initialize the operating point states with the values you obtained from the simulation snapshot.



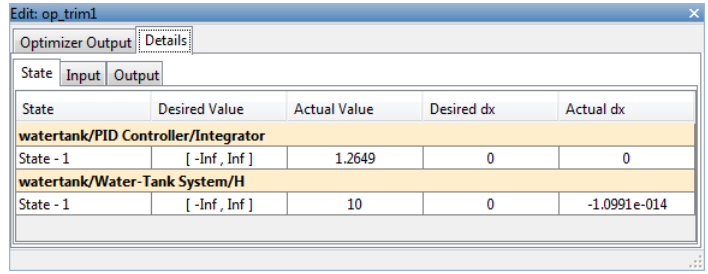
The state values displayed in the Specifications for trim dialog box update to reflect the new values.



8 Click to find the optimized operating point using the states at `t = 10` as the initial values.

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

The **Actual dx** values are near zero. This result indicates that the operating point meets the steady state specifications.



Initialize Operating Point Search (MATLAB Code)

This example show how to use `initopspec` to initialize operating point object values for optimization-based operating point search.

1 Open Simulink model.

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

2 Extract an operating point from simulation after 10 time units.

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

3 Create operating point specification object.

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

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

4 Configure initial values for operating point search.

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

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

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

`opreport` describes the optimization algorithm status at the end of the operating point search.

```
 Operating 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:     -1.1e-014 (0)

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

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

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

Related Examples

- "Steady-State Operating Points from State Specifications" on page 1-16

More About

- "Change Operating Point Search Optimization Settings" on page 1-34
- "Steady-State Operating Point (Trimming)" on page 1-2

Compute Steady-State Operating Points for SimMechanics Models

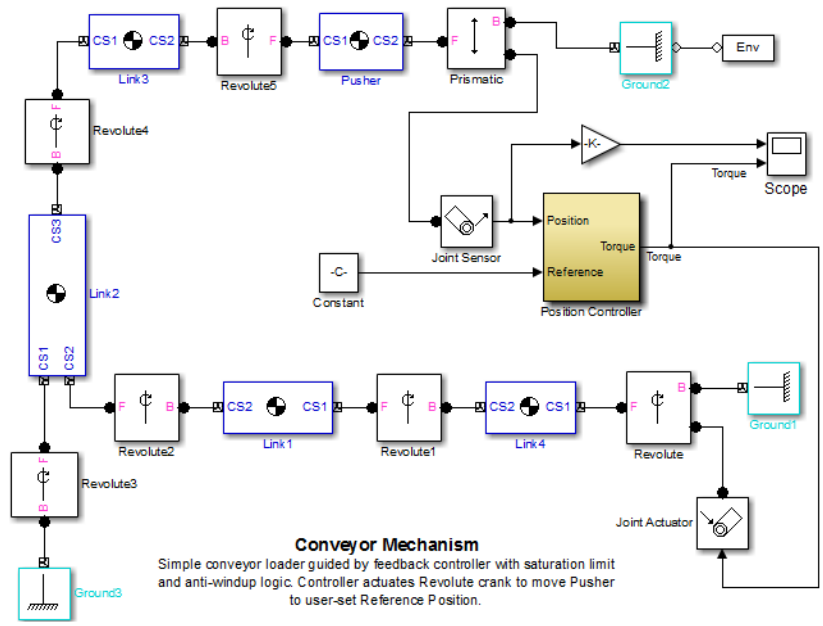
This example shows how to compute the steady-state operating point of a SimMechanics model from specifications.

Note You must have installed SimMechanics software to execute this example on your computer.

---

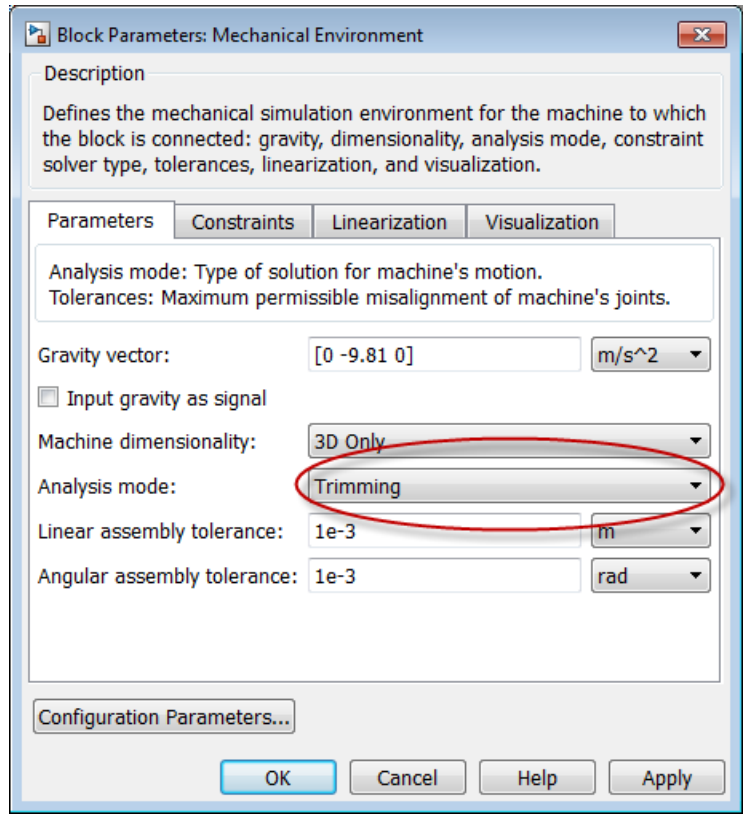
1 Open the SimMechanics model.

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



2 Double-click the `Env` block to open the Block Parameters dialog box.

3 In the **Parameters** tab, select `Trimming` as the **Analysis mode**. Click **OK**.



This action adds an output port to the model with constraints that must be satisfied to a ensure a consistent SimMechanics machine.

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

The Linear Analysis Tool for the model opens.

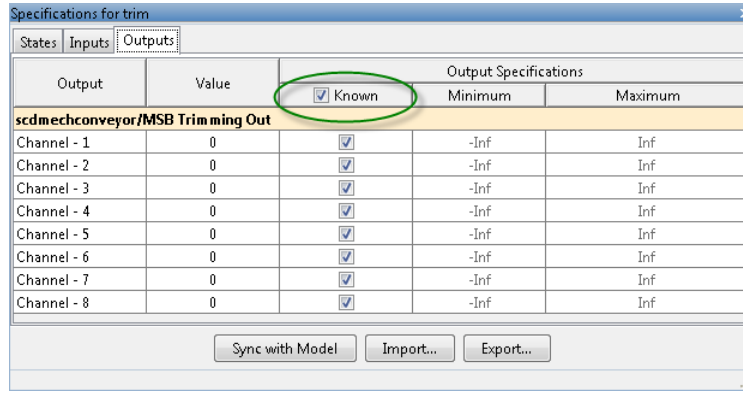
5 In the **Linear Analysis** tab, click **Trim Model**. Then click **Specifications**.

The Specifications for trim dialog box appears.

By default, the software specifies all model states to be at equilibrium (as shown in the **Steady State** column). The **Outputs** tab shows the

error constraints in the system that must be set to zero for steady-state operating point search.

6 In the **Outputs** tab, select **Known** to set all constraints to `0`.



You can now specify additional constraints on the operating point states and input levels, and find the steady-state operating point for this model.

After you finish steady-state operating point search for the SimMechanics model, reset the **Analysis mode** to `Forward dynamics` in the Env block parameters dialog box.

More About

- “Change Operating Point Search Optimization Settings” on page 1-34

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, use the Linear Analysis Tool to interactively configure your operating points search. 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. Open the Linear Analysis Tool for the model.

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

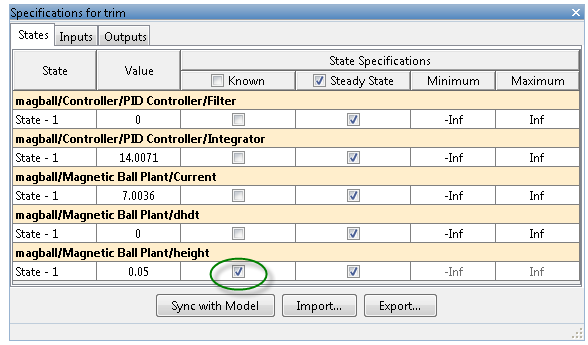
3. Open the Specifications for trim dialog box.

   In the **Linear Analysis** tab, click **Trim Model**. The **Trim Model** tab should open.

   Click **Specifications**.

   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 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 Click **Generate MATLAB Script** in the **Trim** list to automatically generate a MATLAB script.

The MATLAB Editor window opens with the generated script.

7 Edit the script:

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`, you must update the reference ball height, specified by the Desired Height block.

Your script should now look similar to this (excluding most comments):

```
function [op,opreport] = myoperatingpointsearch

%% Specify the model name
sys = 'magball';
load_system(sys)

%% Create operating point specification object
opspec = operspec(sys)

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

%% Loop over height values to find the corresponding steady-state
%% operating points
for ct = 1:numel(height)
    % Set the ball height in the specification
```

```
    opspec.States(5).x = height(ct);

    % Update model parameter
    set_param('magball/Desired Height','Value',num2str(height(ct)));

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

See Also

- `findop`
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177

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. Click **Trim Model** and click **Optimization Options**.

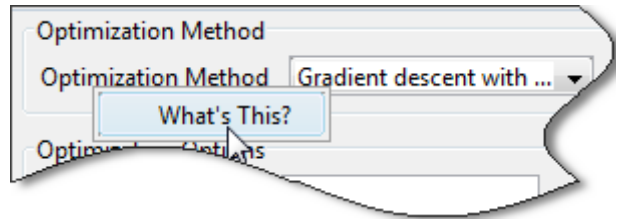
   This action opens the Options for trim dialog box.

2. Change the appropriate optimization 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?**.



Related Examples

- "Steady-State Operating Points from State Specifications" on page 1-16
- "Steady-State Operating Point to Meet Output Specification" on page 1-21
- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-31

More About

- "Steady-State Operating Point (Trimming)" on page 1-2

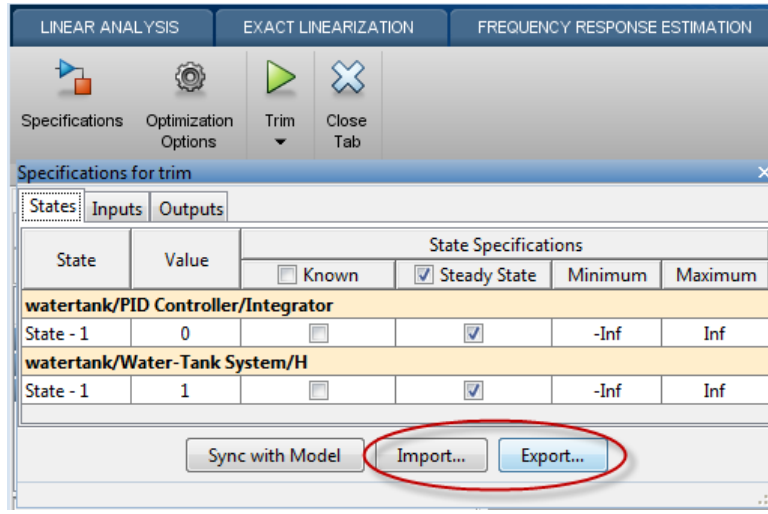
Import and Export Specifications For Operating Point Search

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" on page 1-38.)

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 from it.

To import or save an operating point specification:

1. In the Linear Analysis Tool, on the Linear Analysis Tab, click **Trim Model** to open the Trim Model Tab.
2. Click **Specifications** to open the Specifications for trim dialog box.
3. Click **Import** to import 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.

Batch Compute Steady-State Operating Points

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` a vector 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.

1 Open Simulink model.

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

2 Create operating point specification object.

```
opspec1 = operspec(sys);
```

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

3 Configure the output specification.

```
blk = [sys '/rad//s to rpm'];
opspec1 = addoutputspec(opspec1,blk,1);
opspec1.Outputs(1).Known = true;
opspec1.Outputs(1).y = 1500;
```

`opspec1` specifies a stead-state operating point in which the output of the block `rad/s to rpm` is fixed at 500.

Note Alternatively, you can configure an operating point specification using the Linear Analysis Tool and export the specification to the MATLAB workspace. See “Import and Export Specifications For Operating Point Search” on page 1-36 for more information.

---

4 Create and configure additional operating point specifications.

```
opspec2 = copy(opspec1);
opspec2.Outputs(1).y = 2000;

opspec3 = copy(opspec1);
opspec3.Outputs(1).y = 2500;
```

Using the `copy` command creates an independent operating point specification that you can edit without changing `opspec1`. Here, the specifications `opspec2` and `opspec3` are identical to `opspec1`, except for the target output level.

5 Find the operating points that meet each of the three output specifications.

```
opspecs = [opspec1,opspec2,opspec3];
ops = findop(sys,opspecs);
bdclose(sys);
```

Pass the three operating point specifications to `findop` in the vector `opspecs`. When you give `findop` a vector of operating point specifications, it finds all the operating points with only one model compilation. `ops` is a vector of operating point objects for the model `scdspeed` that correspond to the three specifications in the vector.

See Also

`findop`

Related Examples

- "Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code" on page 1-31
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177

Steady-State Operating Points from Simulation

In this section...

"Simulation Snapshot Operating Points" on page 1-40

"Compute Operating Points at Simulation Snapshots" on page 1-40

Simulation Snapshot Operating Points

You can compute a steady-state operating point (or equilibrium operating point) from model simulation. The resulting operating point consists of the state values and model input levels at the specified simulation time.

Simulation-based operating point computation requires that you configure your model by specifying:

- Initial conditions that cause your model to converge to equilibrium
- Simulation time at which the model reaches equilibrium

You can use the simulation snapshot operating point to initialize the trim point 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.

Compute Operating Points at Simulation Snapshots

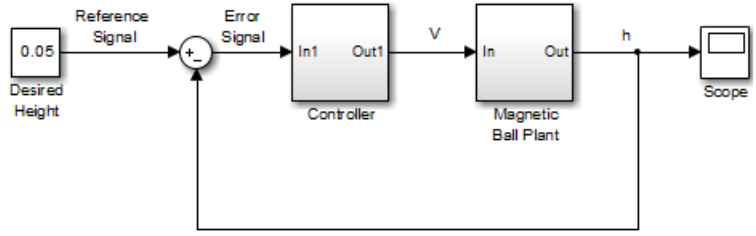
This example shows how to use the Linear Analysis Tool to compute an operating point at specified simulation times (or *simulation snapshots*).

Code Alternative

Use `findop` to compute operating point at simulation snapshot. For examples and additional information, see the `findop` reference page.

1 Open Simulink model.

```
sys = 'magball';
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, click the **Operating Point Snapshots** tab.

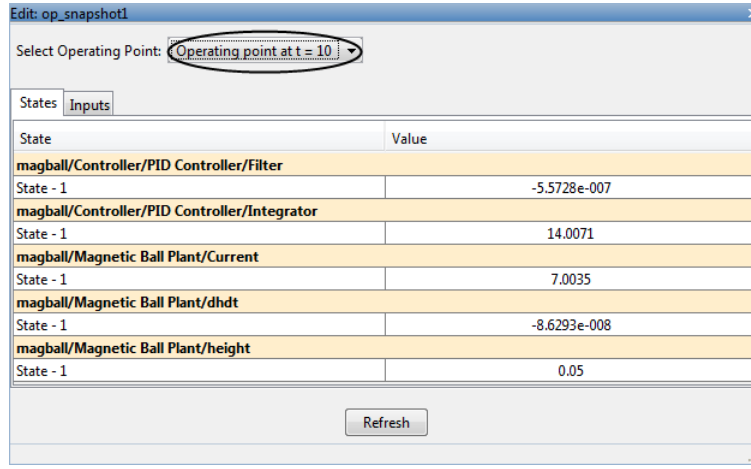
4 Specify `[1,10]` in the **Simulation Snapshot Times** field. Press **Enter**.

This vector specifies operating points at `t = 1` and `t = 10`.

5 Click to take a snapshot of the system at the specified times.

A new variable, `op_snapshot1`, appears in the **Linear Analysis Workspace**. `op_snapshot1` contains the two operating points.

6 Double-click `op_snapshot1` to see the resulting operating points. Select an operating point of interest from the **Select Operating Point** list to see it.



For example, to evaluate your operating point from a simulation snapshot:

1. Initialize the model at the operating point (see "Simulate Simulink Model at Specific Operating Point" on page 1-43)
2. Add Scope blocks to show the output signals that should reach steady state during the simulation.
3. Run the simulation to check whether these key signals are at steady state.

More About

- "Choosing Between Simulation Snapshot and Operating Point from Specifications" on page 1-12

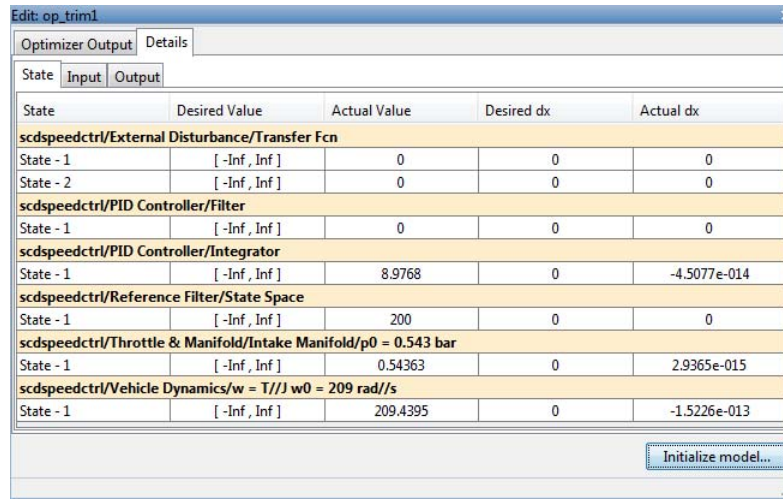
Simulate Simulink Model at Specific Operating Point

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

1. Compute a steady-state operating point, as described in "Compute Operating Points at Simulation Snapshots" on page 1-40.

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

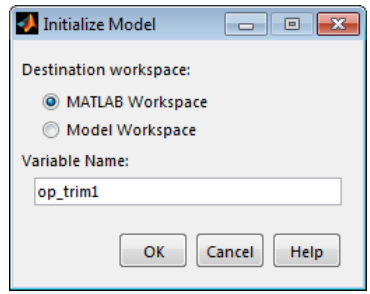
   The Edit dialog box opens.



State	Desired Value	Actual Value	Desired dx	Actual dx
scdspeedctrl/External Disturbance/Transfer Fcn				
State - 1	[-Inf , Inf]	0	0	0
State - 2	[-Inf , Inf]	0	0	0
scdspeedctrl/PID Controller/Filter				
State - 1	[-Inf , Inf]	0	0	0
scdspeedctrl/PID Controller/Integrator				
State - 1	[-Inf , Inf]	8.9768	0	-4.5077e-014
scdspeedctrl/Reference Filter/State Space				
State - 1	[-Inf , Inf]	200	0	0
scdspeedctrl/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar				
State - 1	[-Inf , Inf]	0.54363	0	2.9365e-015
scdspeedctrl/Vehicle Dynamics/w = T//J w0 = 209 rad//s				
State - 1	[-Inf , Inf]	209.4395	0	-1.5226e-013

3. Click **Initialize model**.

   The Initialize Model dialog box opens.



4 Use the default **Variable Name** for the operating point object. Alternatively, you can edit this 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 uses this operating point as initial conditions 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.

Related Examples

- “Steady-State Operating Points from State Specifications” on page 1-16
- “Compute Operating Points at Simulation Snapshots” on page 1-40

Handling Blocks with Internal State Representation

In this section...

"Operating Point Object Excludes Blocks with Internal States" on page 1-45

"Identifying Blocks with Internal States in Your Model" on page 1-46

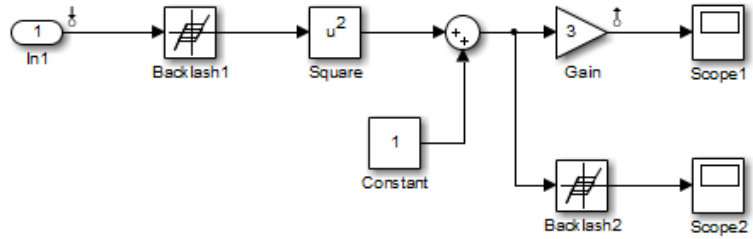
"Configuring Blocks with Internal States for Steady-State Operating Point Search" on page 1-46

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 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 this 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')
```

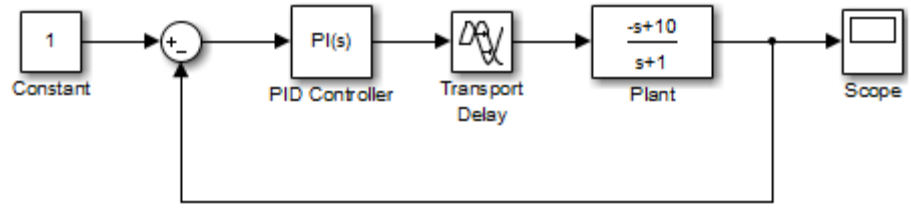
where `sys` is your model, specified as a string. This command also returns the number of occurrences of each block.

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 issues for Memory blocks, 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 1 during 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.



To fix this issue, select the **Direct feedthrough of input during linearization** option in the Block Parameters dialog box before searching for an operating point. This setting lets the PID Controller block push a nonzero value to the Plant block.

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.

You can also set direct feedthrough options at the command-line instead of using the block parameter dialog box.

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

Synchronize Simulink Model Changes with Operating Point Specifications

In this section...

"Synchronize Simulink Model Changes with Linear Analysis Tool" on page 1-48

"Synchronize Simulink Model Changes with Existing Operating Point Specification Object" on page 1-51

Synchronize Simulink Model Changes with Linear Analysis Tool

This example shows how to update the operating point specifications in the Linear Analysis Tool to reflect changes to the Simulink model.

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

1. Open Simulink model.

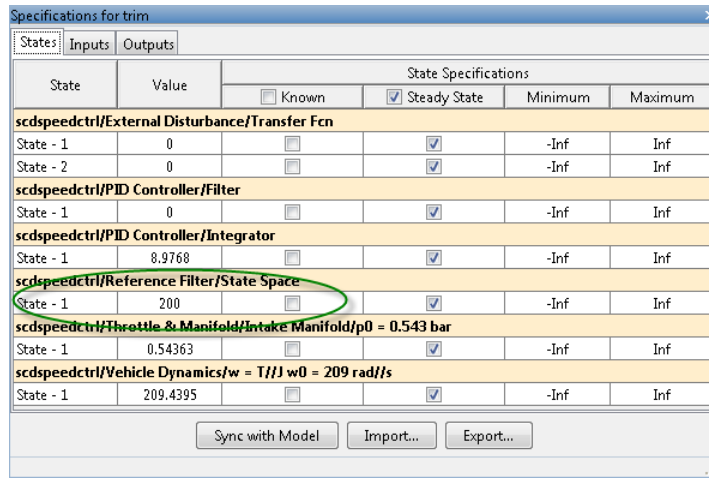
   ```
   sys = ('scdspeedctrl');
   open_system(sys)
   ```

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.

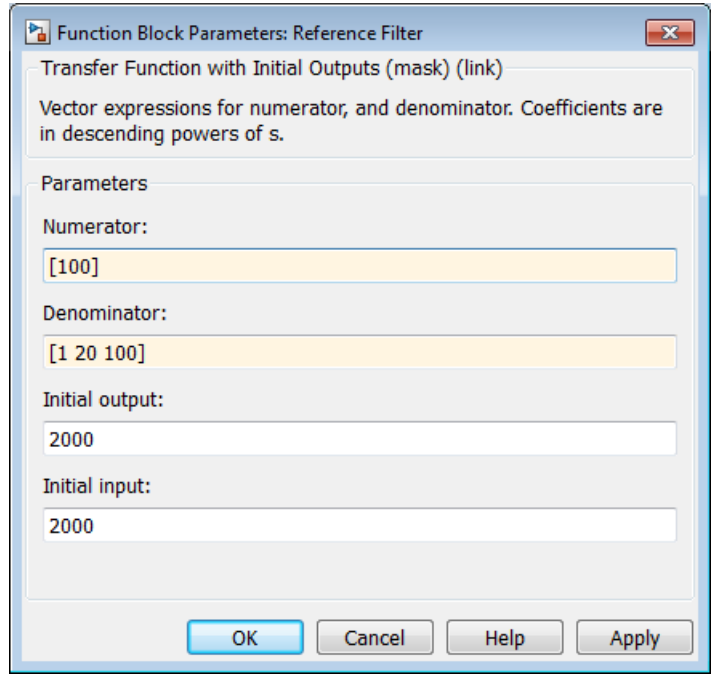
3. In the **Linear Analysis** tab, click **Trim Model**. Then click **Specifications**.

   The Specifications for trim dialog box appears.



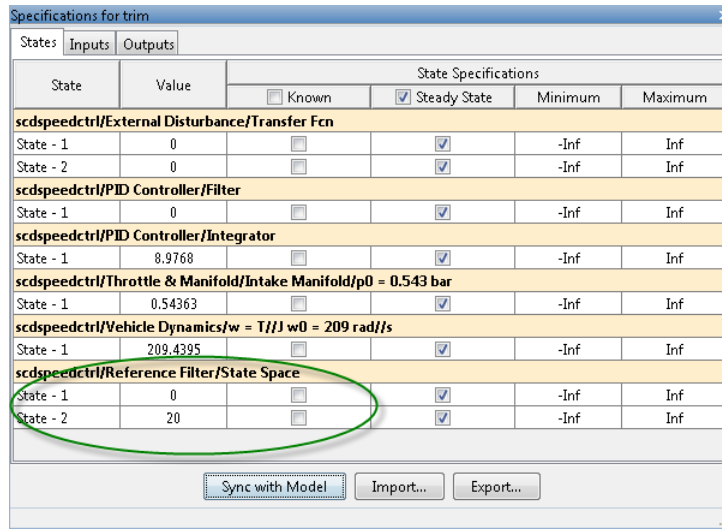
The Reference Filter block contains just one state.

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 adds a state to the Simulink model.

5 In the Specifications for trim dialog, click **Sync with Model** to synchronize the operating point specifications in the Linear Analysis Tool with the model.



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

6 Click to compute the operating point.

Synchronize Simulink Model Changes with Existing Operating Point Specification Object

This example shows how to use `update` to update the operating point specification object after you update the Simulink model.

1 Open Simulink model.

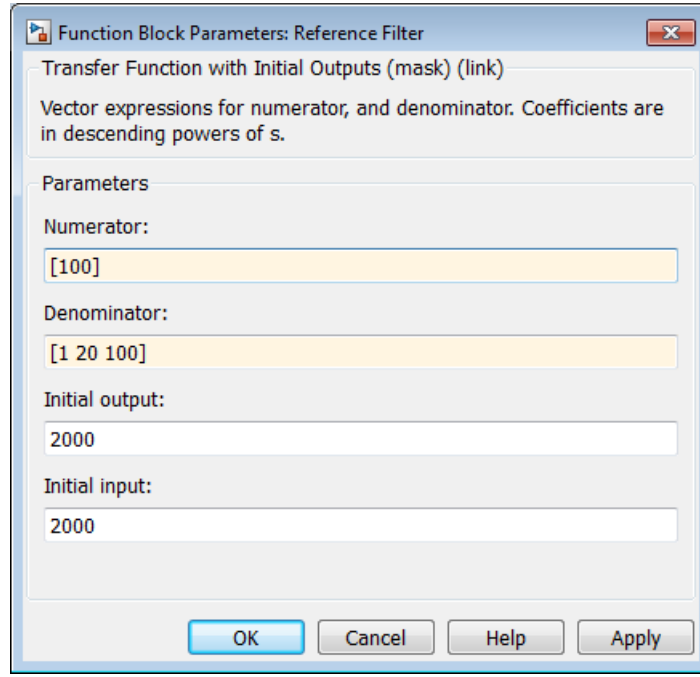
```
sys = 'scdspeedctrl';
open_system(sys);
```

2 Create operating point specification object.

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

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

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 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);
```

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

2

Linearization

- "Linearizing Nonlinear Models" on page 2-3
- "Specify Model Portion to Linearize" on page 2-13
- "Plant Linearization" on page 2-33
- "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37
- "Linearize at Model Operating Point" on page 2-43
- "Linearize at Trimmed Operating Point" on page 2-57
- "Linearize at Simulation Snapshots and Triggered Events" on page 2-62
- "Ordering States in Linearized Model" on page 2-81
- "Time-Domain Validation of Linearization" on page 2-86
- "Frequency-Domain Validation of Linearization" on page 2-90
- "Visualize Models" on page 2-96
- "Generate MATLAB Code for Linear Analysis Tool Session" on page 2-99
- "Troubleshooting Linearization" on page 2-101
- "Controlling Block Linearization" on page 2-117
- "Models with Pulse Width Modulation (PWM) Signals" on page 2-140
- "Specifying Linearization for Model Components Using System Identification" on page 2-142
- "Speeding Up Linearization of Complex Models" on page 2-150
- "Exact Linearization Algorithm" on page 2-152
- "How the Software Treats Loop Openings" on page 2-158

- "What Is Batch Linearization?" on page 2-160
- "Choosing Batch Linearization Tools" on page 2-162
- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164
- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183
- "Specify Parameter Samples" on page 2-190
- "Validate Batch Linearization Results" on page 2-196

Linearizing Nonlinear Models

In this section...

"What Is Linearization?" on page 2-3

"Applications of Linearization" on page 2-5

"Linearization in Simulink® Control Design™" on page 2-6

"Choosing Linearization Tools" on page 2-7

"Model Requirements for Exact Linearization" on page 2-10

"Operating Point Impact on Linearization" on page 2-11

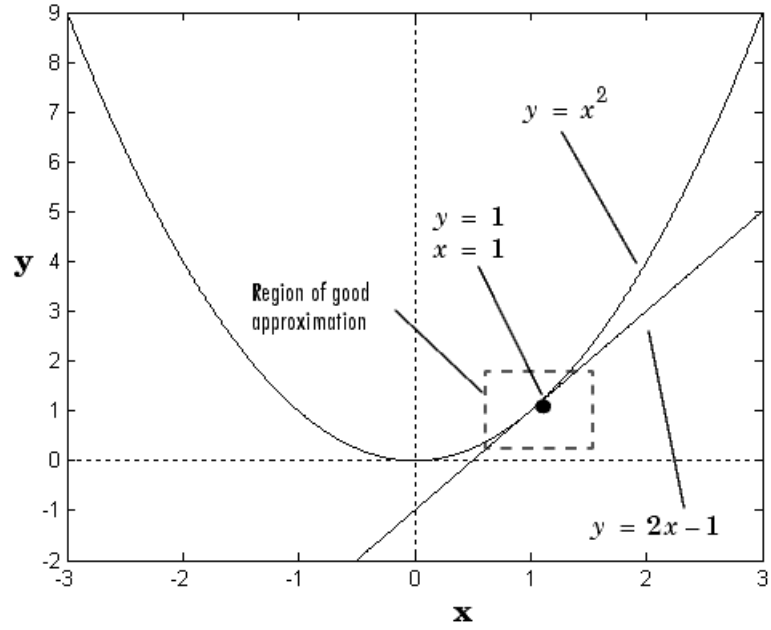
What Is Linearization?

Linearization is a linear approximation of a nonlinear system that is valid in a small region around the 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:

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

Examples and How To

- "Linearize Simulink Model" on page 2-43
- "Plant Linearization" on page 2-33
- "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

More About

- "Applications of Linearization" on page 2-5
- "Choosing Linearization Tools" on page 2-7

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 "Choosing a Compensator Design Approach" on page 4-2.

- 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 (requires Robust Control Toolbox™).

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 "Controlling Block Linearization" on page 2-117.
- Simulink Control Design automatically removes nonminimal states.
- Ability to specify linearization to be uncertain (requires Robust Control Toolbox)

More About

"Exact Linearization Algorithm" on page 2-152

Choosing Linearization Tools

- "Choosing Simulink® Control Design™ Linearization Tools" on page 2-7
- "Choosing Exact Linearization Versus Frequency Response Estimation" on page 2-8
- "Linearization Using Simulink® Control Design™ Versus Simulink" on page 2-9

Choosing Simulink Control Design Linearization Tools

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

Linearization Tool	When to Use
Linear Analysis Tool	• Interactively explore Simulink model linearization under different operating conditions. • Diagnose linearization issues. • 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.

Linearization Tool	When to Use
`slLinearizer`	Batch linearize for varying model parameter values, operating points, and I/O sets.
Linear Analysis Plots blocks	• 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, you should use exact linearization instead of frequency response estimation to obtaining a linear approximation of a Simulink model.

Exact linearization:

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

  Frequency response estimation returns frequency response data. To create a transfer function or a state-space model from the resulting frequency response data requires an extra step using System Identification Toolbox™ to fit a model.

Use frequency response estimation:

- To validate exact linearization accuracy.
- When your Simulink model contains discontinuities or non-periodic event-based dynamics.
- To study the impact of amplitude size on frequency response.

  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" on page 2-43.	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 2-57.	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 "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37.	Yes, but requires that you delete feedback signals in your model to open the loop

	Simulink Control Design Linearization	Simulink Linearization
Control linear model state ordering	Yes See "Ordering States in Linearized Model" on page 2-81.	No
Control linearization of individual blocks	Yes. Lets you specify custom linearization behavior for both blocks and subsystems. See "Controlling Block Linearization" on page 2-117.	No
Linearization diagnostics	Yes. Identifies problematic blocks and lets you examine the linearization value of each block. See "Linearization Troubleshooting Overview" on page 2-101.	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

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.

More About

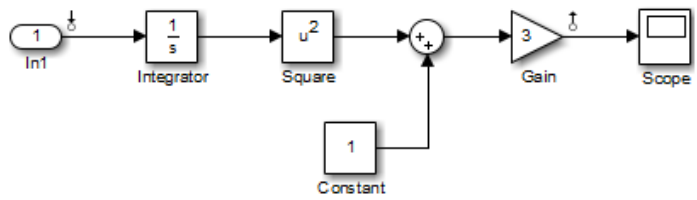
"Exact Linearization Algorithm" on page 2-152

Operating Point Impact on Linearization

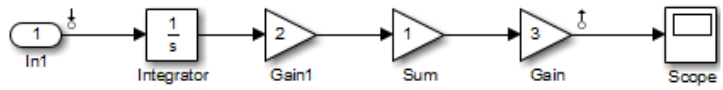
Choosing the right operating point for linearization is critical for obtaining an accurate linear model. The linear model is an approximation of the nonlinear model that is valid only near the operating point at which you linearize the model.

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

Specify Model Portion to Linearize

In this section...

"Specifying Subsystem, Loop, or Block to Linearize" on page 2-13
"Opening Feedback Loops" on page 2-14
"Ways to Specify Portion of Model to Linearize" on page 2-16
"Specify Portion of Model to Linearize in Simulink Model" on page 2-17
"Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-20
"Edit Portion of Model to Linearize in Linear Analysis Tool" on page 2-25
"Select Bus Elements as Linear Analysis Points" on page 2-27

Specifying Subsystem, Loop, or Block to Linearize

Simulink Control Design lets you specify the subsystem, loop, or block to linearize using linearization input and output points (linearization I/O points).

A *linearization input point* defines the additive input signal to the linear model. A *linearization output point* defines the output signal of the linear model.

You can linearize:

- Closed- or open-loop responses using a linearization input point on the input signal to the portion of the model you want to linearize, and a linearization output point at the output signal of that portion of the model.
- Specific subsystem or block.

  In this case, linearization I/O points are the input and output signals corresponding to the subsystem or block.

You can define other linear models using additional types of linear analysis points:

- **Loop Transfer** — Specifies an output point before a loop opening followed by an input. Use this input/output type to compute the open-loop transfer function around the loop.

- **Loop Break** — Specifies a loop opening. Use to compute open-loop transfer function around a loop. Typically, you use this input/output type when you have nested loops or to ignore the effect of some loops.
- **Sensitivity** — Specifies an additive input followed by an output measurement. Use to compute sensitivity transfer function for an additive disturbance at the signal.
- **Complementary Sensitivity** — Specifies an output followed by an additive input. Use to compute closed-loop transfer function around the loop.

Linearization I/O points are pure annotations and do not impact model simulation.

More About

"Ways to Specify Portion of Model to Linearize" on page 2-16

Opening Feedback Loops

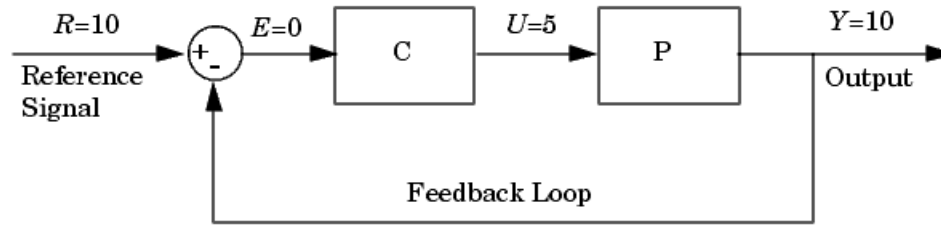
If your model contains one or more feedback loops, you can choose to linearize an open-loop or a closed-loop system.

Simulink Control Design lets you remove the effects of the feedback loop by inserting an *open loop point* without having to manually break signal lines. In fact, for nonlinear models, do not open the loop by manually removing the feedback signal from the model; this action changes the model operating point and produces a different linear model.

Note If a model is already linear, it has the same form regardless of the operating point.

Correct placement of the loop opening is critical to obtaining the right linear model. For example, you might want to linearize only the plant model in a feedback control loop.

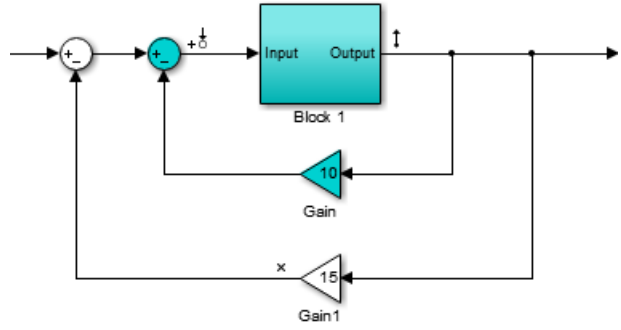
To understand the difference between open-loop and closed-loop analysis, consider this single-loop control system.



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

To linearize only the plant P, you must open the loop at the output of the P block. If you do not open the loop, and if C and P are linear, the linearized model between U and Y is $\frac{P(s)}{1+P(s)C(s)}$.

The loop opening does not need to be in the same location as the linearization input or output point. For example, the next figure shows a loop opening after the gain on the outer feedback loop, which removes the effect of this loop from the linearization. To check whether you correctly excluded the feedback signal, linearize the model and highlight the blocks included in the linearization.



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

More About

- "Ways to Specify Portion of Model to Linearize" on page 2-16
- "Highlighting Linearized Blocks" on page 2-108
- "How the Software Treats Loop Openings" on page 2-158

Ways to Specify Portion of Model to Linearize

There are several ways to specify linearization inputs, outputs, and loop-opening locations (*linear analysis points*, *linearization I/O sets*, or, simply, *I/O sets*) that define the portion of the model you want to linearize. Each method has its own advantages. You can:

- Specify linearization I/O points and loop openings directly in the model. An advantage of this method is that the locations of linearization I/O points and loop openings are shown graphically in the model. When you specify linearization I/O sets this way and save the model, the I/O set persists in the model.
- Interactively define linearization I/O sets using the Linear Analysis Tool. The **Create linearization I/O set** dialog box in the Linear Analysis Tool allows you to define multiple open-loop or closed-loop transfer functions for your model interactively. This approach does not make changes to the model.
- Define linearization I/O sets at the command line using `linio`. This method allows you to define multiple open-loop or closed-loop transfer functions without changing the model.
- Define analysis points and openings for an `slLinearizer` interface at the command line. This method allows you to obtain multiple open-loop or closed-loop transfer functions without changing the model.

Specify Portion of Model to Linearize in Simulink Model

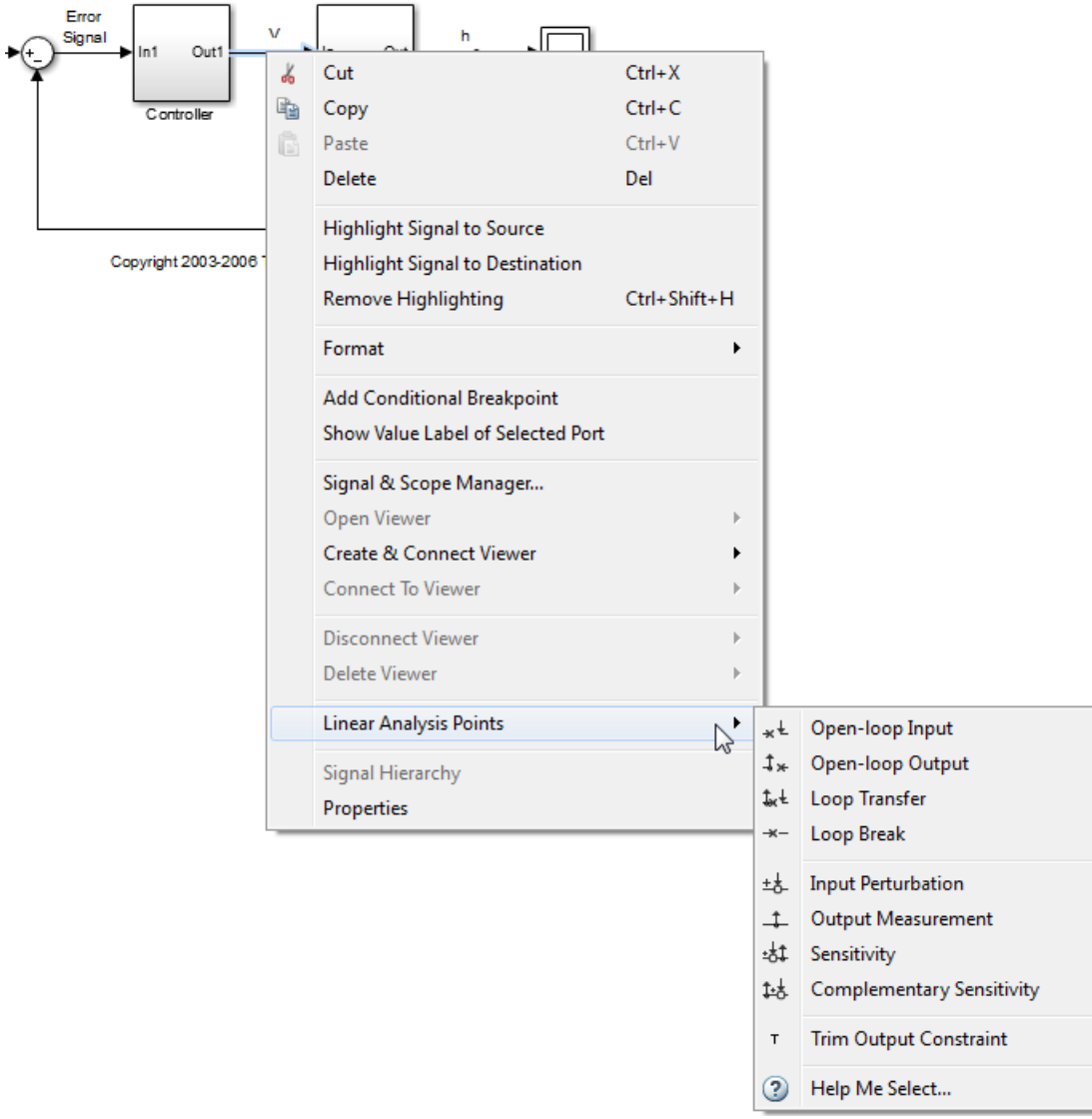
To specify linearization I/O points and loop openings directly in your Simulink model:

1. Right-click on the signal you want to define as a linearization input point or output point.

   This action opens a context menu on the signal.

2. Hover the cursor over **Linear Analysis Points** in the context menu.

   A submenu appears listing types of linear analysis points.



3 Select the type of linear analysis point you want to define at the signal.

a Right-click the block output signal corresponding to the linearization input point. For example, select **Linear Analysis Points > Input**

Perturbation. This type of input point specifies an additive input to a signal.

If you want to specify the signal as a open-loop input point, select **Linear Analysis Points > Open-loop Input**. This option specifies an input point after a loop opening. Opening the loop removes the effects of the feedback signal on the linearization without changing the model operating point. The loop opening marker appears in the model.

Caution Do not open the loop by manually removing the feedback signal from the model. Removing the signal manually changes the operating point of the model.

---

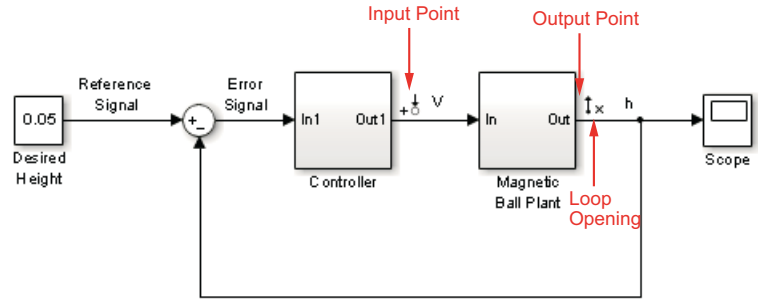
b Right-click the block output signal corresponding to the linearization output point. For example, select **Linear Analysis Points > Output Measurement**. This type of output point takes measurement at a signal.

If you want to specify the signal as a open-loop output point, select **Linear Analysis Points > Open-loop Output**. This option specifies an output point before a loop opening.

Depending on the response you want, you can select one of the following additional linear analysis points:

- **Loop Transfer** — Specifies an output point before a loop opening followed by an input. Use this input/output type to compute the open-loop transfer function around the loop.
- **Loop Break** — Specifies a loop opening. Use to compute open-loop transfer function around a loop. Typically, you use this input/output type when you have nested loops or to ignore the effect of some loops.
- **Sensitivity** — Specifies an additive input followed by an output measurement. Use to compute sensitivity transfer function for an additive disturbance at the signal.
- **Complementary Sensitivity** — Specifies an output followed by an additive input. Use to compute closed-loop transfer function around the loop.

When you specify linearization inputs and outputs or loop openings, markers appear in your model indicating the linear analysis point type.



4 Repeat step 3 for all signals you want to define as linearization I/O points or open-loop points.

Specifying linear analysis points using the context menu changes the model (makes the model "dirty", that is, saving the model stores the points with the model).

More About

- "Specifying Subsystem, Loop, or Block to Linearize" on page 2-13
- "Opening Feedback Loops" on page 2-14
- "Ways to Specify Portion of Model to Linearize" on page 2-16

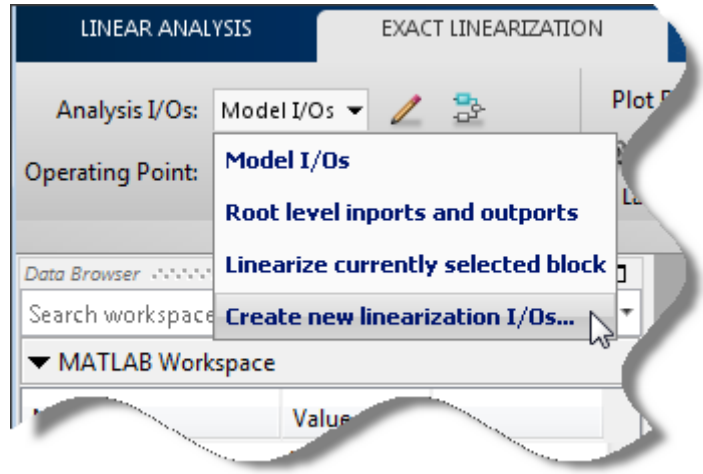
Specify Portion of Model to Linearize in Linear Analysis Tool

You use linearization inputs, outputs, and loop-opening locations (*linearization I/O sets*) to specify which portion of the model to linearize. You can specify one or more linearization I/O sets interactively in the Linear Analysis Tool, without introducing changes to the model.

To access the Create linearization I/O set dialog box:

1 Click the Exact Linearization or Frequency Response Estimation tab.

2 From the **Analysis I/Os** drop-down list, select `Create new linearization I/Os`.



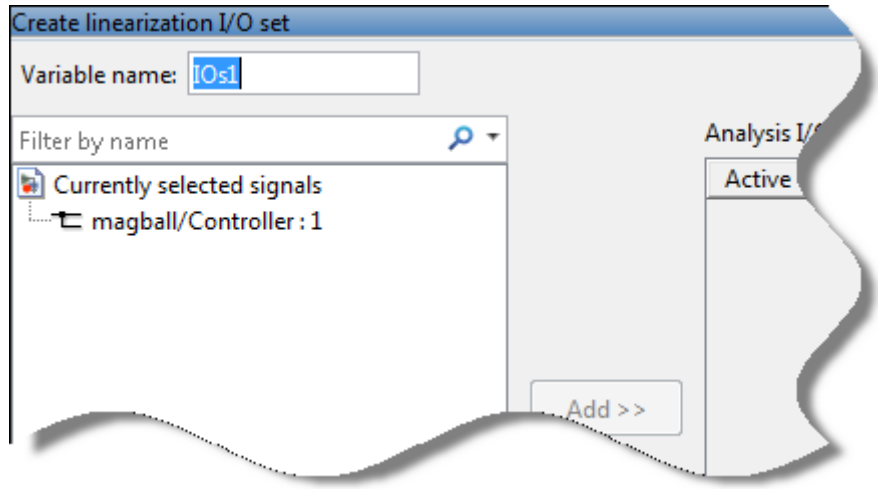
The Create linearization I/O set dialog box opens.

Create Linearization I/O Set

To create a new linearization I/O set:

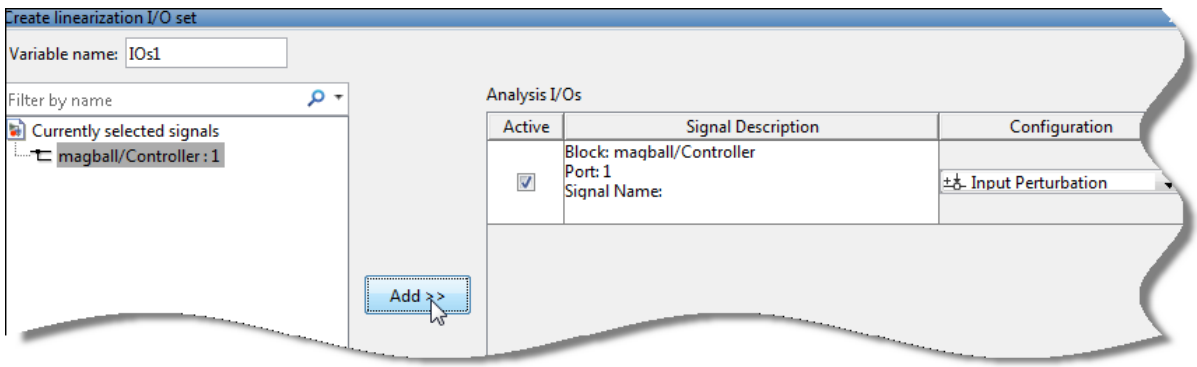
1 In your Simulink model, select a signal that you want to define as a linearization input or output point.

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



2 In the Create linearization I/O set dialog box, click the signal name under Currently selected signals.

3 Click **Add**. The signal appears in the list of Analysis I/Os.



4 Select the linearization point type from the **Configuration** drop-down list. For example:

- If you want the selected signal to be a linearization output point, select `Output Measurement`.

- If you want the signal to be an open-loop output point, select `Open-loop Output`.

5. Repeat steps 1–4 for any other signals you want to define as linearization I/O points.

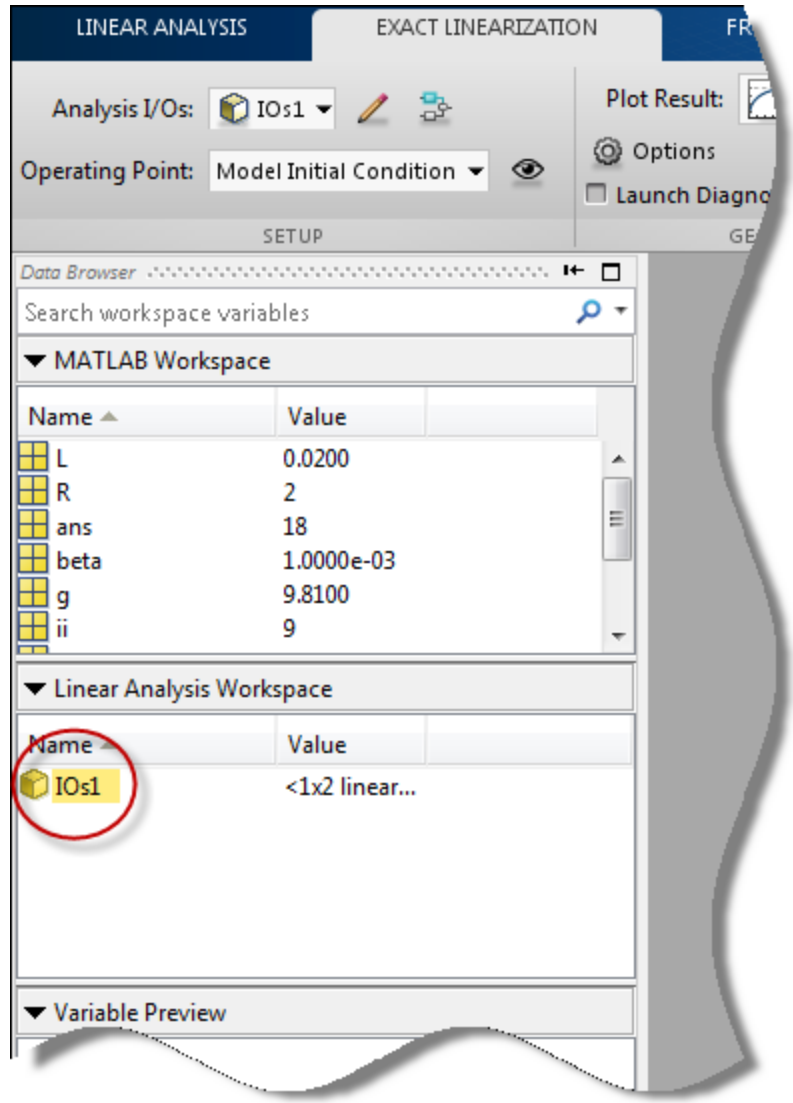
Tip To highlight in the Simulink model the blocks that are included in the linearization specified by the current list of Analysis I/Os, click **Highlight**.

---

6. After you define all the signals for the I/O set, enter a name for the I/O set in the **Variable name** field located at the top-left of the window.

7. Click **OK**.

The Create linearization I/O set dialog box closes. A new linearization I/O set appears in the Linear Analysis Workspace of the Linear Analysis Tool. The new linearization I/O set displays the name you specified.



The newly created linearization I/O set is automatically selected in the **Analysis I/Os** menu for either the Exact Linearization or Frequency Response Estimation tab, depending on which you selected originally.

Creating linearization I/O sets in the Linear Analysis Tool does not change the Simulink model. You can create multiple I/O sets for a single model.

More About

- "Specifying Subsystem, Loop, or Block to Linearize" on page 2-13
- "Opening Feedback Loops" on page 2-14
- "Ways to Specify Portion of Model to Linearize" on page 2-16

Edit Portion of Model to Linearize in Linear Analysis Tool

You can interactively edit a linearization I/O set stored in the Linear Analysis Workspace using the Linear Analysis Tool Edit dialog box.

Open Edit Dialog Box

To open the Edit dialog box for editing a linearization set, either:

- In the Linear Analysis Workspace, double-click the I/O set.
- Click either the Exact Linearization or Frequency Response Estimation tab. If the I/O set is selected in the **Analysis I/Os** menu, click the edit icon next to the **Analysis I/Os** menu.

Either of these actions opens the I/O set edit dialog box for the linearization I/O set. You can now edit the I/O set as needed. When you have finished editing, click to close the dialog box and save your changes.

Tip To highlight in the Simulink model the blocks that are included in the linearization specified by the current list of Analysis I/Os, click **Highlight**.

Add Signal to I/O Set

To add a linearization input point, output point, or loop opening to the linearization I/O set:

1 In your Simulink model, select a signal that you want to add to the linearization I/O set.

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

2 In the Create linearization I/O set dialog box, click the signal name under Currently selected signals.

3 Click **Add**. The signal appears in the list of Analysis I/Os.

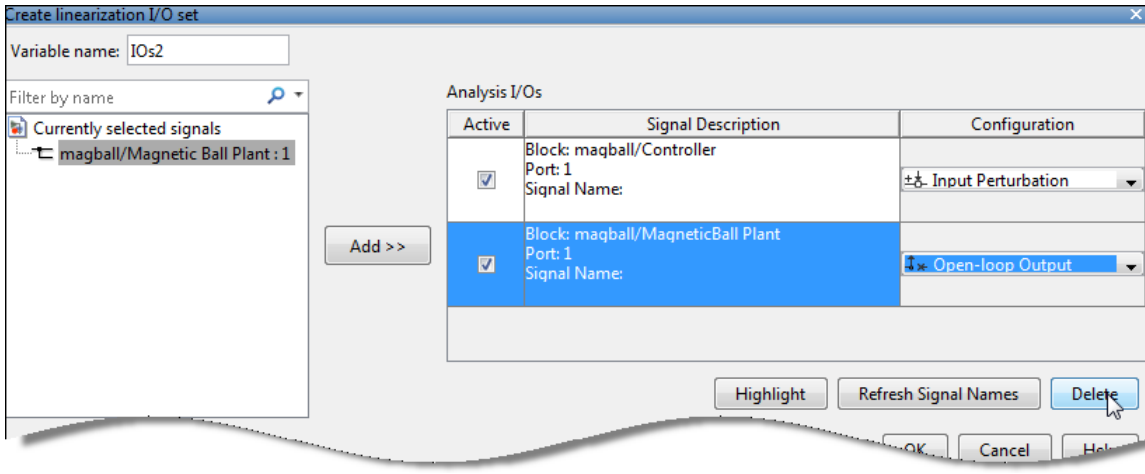
4 Select the linear analysis point type from the **Configuration** drop-down list. For example:

   - If you want the selected signal to be a linearization output point, select `Output Measurement`.
   - If you want the signal to be an open-loop linearization output point, select `Open-loop Output`.

Remove Signal from I/O Set

To remove a linearization input point, output point, or loop opening from the linearization I/O set:

1 Select the signal in the list of Analysis I/Os.



2 Click **Delete** to remove the signal from the linearization I/O set.

Change Linear Analysis Point Type

To change the type of linear analysis point type for a signal, locate the signal in the list of Analysis I/Os. Then, use the **Configuration** drop-down list for the signal to define the type of linear analysis point.

For example, if you want the signal to be a linearization output point, select `Output Measurement` from the **Configuration** drop-down list. If you want the signal to be an open loop output point, select `Open-loop Output`.

Select Bus Elements as Linear Analysis Points

This example shows how to select individual elements in a bus signal as linearization input/output (I/O) points. Linearization I/O points define the portion of the model to linearize.

Code Alternative

Use `linio` to specify model signals as linearization I/O points and loop openings. For examples and additional information, see the `linio` reference page.

1 Open Simulink model.

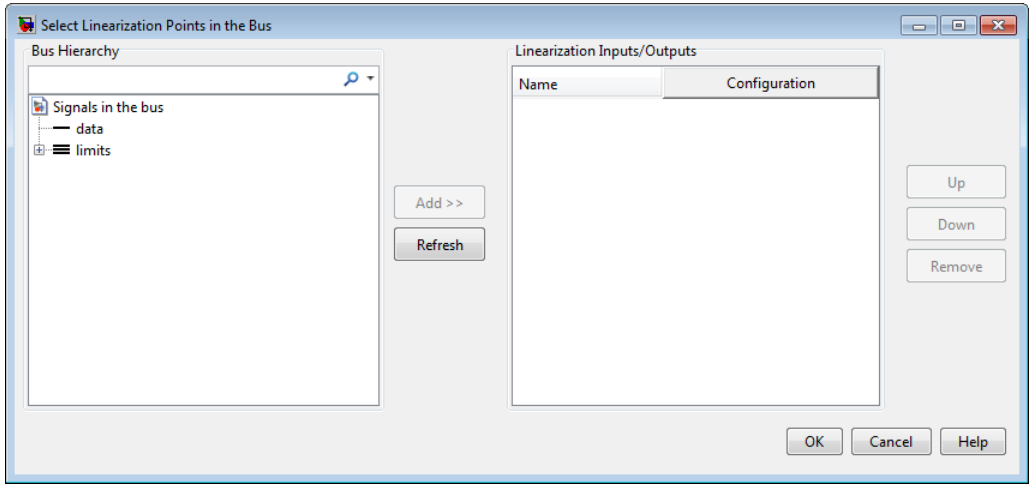
```
sys = 'scdbusselection';
open_system(sys)
```

2 In the Simulink model window, define portion of the model to linearize:

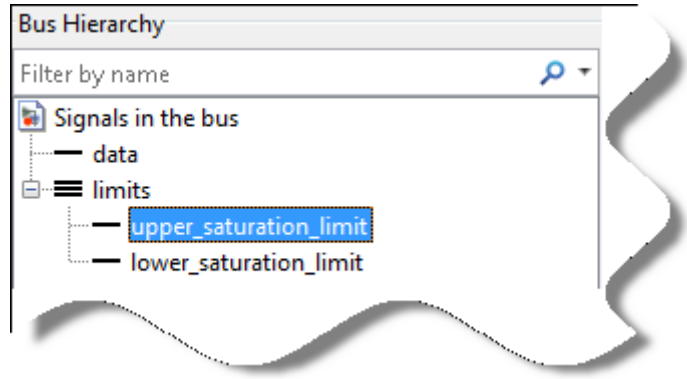
a Right-click the COUNTERBUS signal, and select **Linear Analysis Points > Select Bus Element**.

This option appears only if **Mux blocks used to create bus signals** in the **Configuration Parameters > Diagnostics > Connectivity** pane is `error`. Otherwise, right-clicking the bus signal lets you specify *all* elements in the bus as linearization input or output points.

The Select Linearization Points in the Bus dialog box opens, which shows signals contained in the COUNTERBUS bus signal.



b In the **Bus Hierarchy** area, expand the bus named `limits`. Then, select `upper_saturation_limit`.



Tip For large buses, 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 a MATLAB regular expression.

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

Filtering Options

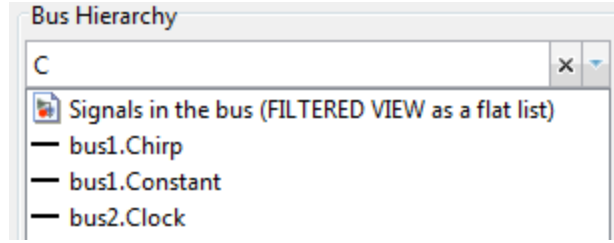
- **Enable regular expression**

  MATLAB regular expression 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**

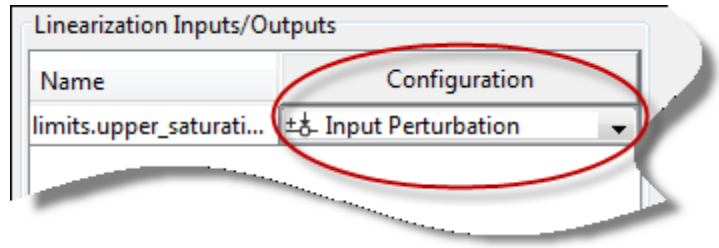
  Flat list format to display the list of filtered signals.

  By default, filtered signals are displayed using a tree format. The flat list format uses dot notation to reflect the hierarchy of bus signals.



a Click **Add**.

The selected signal now appears in the **Linearization Inputs/Outputs** area, and is configured as a linearization input point.



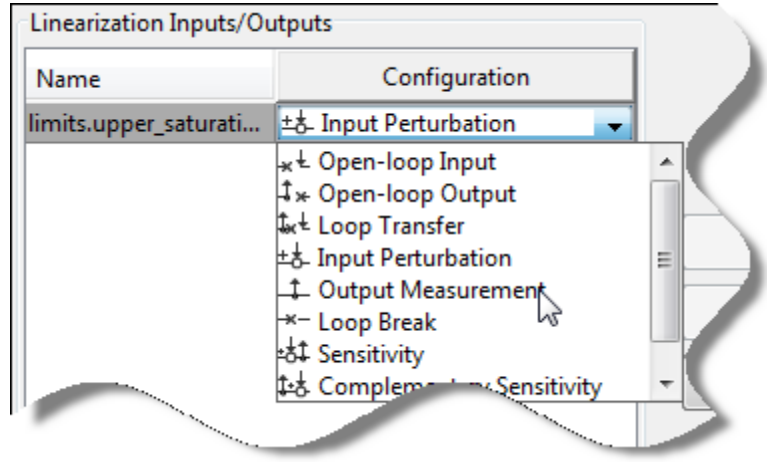
Click **OK**.

b In the Simulink model window, right-click the OUTPUTBUS signal, and select **Linear Analysis Points > Select Bus Element**.

c In the **Bus Hierarchy** area, expand the bus named `limits`, and select `upper_saturation_limit`.

d Click **Add** to add the selected signal to the **Linearization Inputs/Outputs** area.

e Select `Output Measurement` in the **Configuration** column.



Click **OK**.

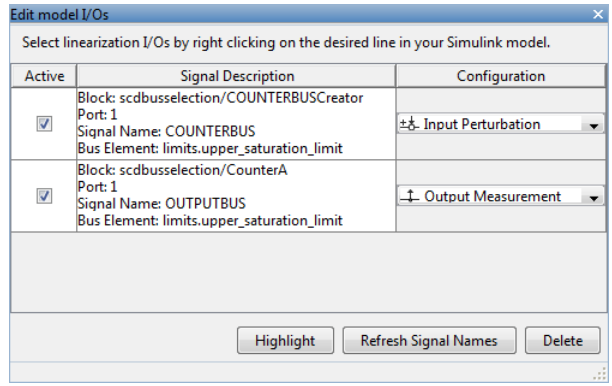
Tip In the Simulink model window, select **Display > Signals & Ports > Linearization Indicators** to view the linearization I/O markers.

You can select multiple elements in the same bus with different I/O types. The marker appears on the bus signal to indicate multiple bus element selections with different I/O types.

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

The Linear Analysis Tool for the model opens.

Click the **Exact Linearization** tab. Click adjacent to the **Analysis I/Os** list to see the bus elements selected as linearization I/O points.



4 In the **Exact Linearization** tab, click to linearize the model using the model initial condition as the operating point.

Related Examples

- “Linearize Simulink Model” on page 2-43
- “Plant Linearization” on page 2-33

- "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37

Plant Linearization

This example shows how to use the Linear Analysis Tool to linearize a plant subsystem at the model operating point. The model operating point consists of the model initial state values and input signals.

Use this simpler approach instead of defining linearization I/O points when the plant is a subsystem or a block.

Code Alternative

Use `linearize`. For examples and additional information, see the `linearize` reference page.

1. Open a Simulink model.

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

2. Open the Linear Analysis Tool.

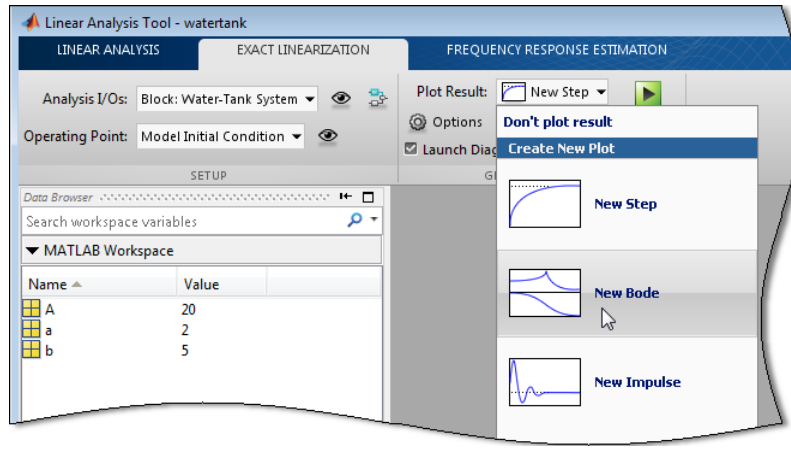
   In the Simulink model window, right-click the Water-Tank System block and select **Linear Analysis > Linearize Block**.

3. View the block selected for linearization.

   Click adjacent to the **Analysis I/Os** drop-down menu.

   The `Water Tank System` block is highlighted.

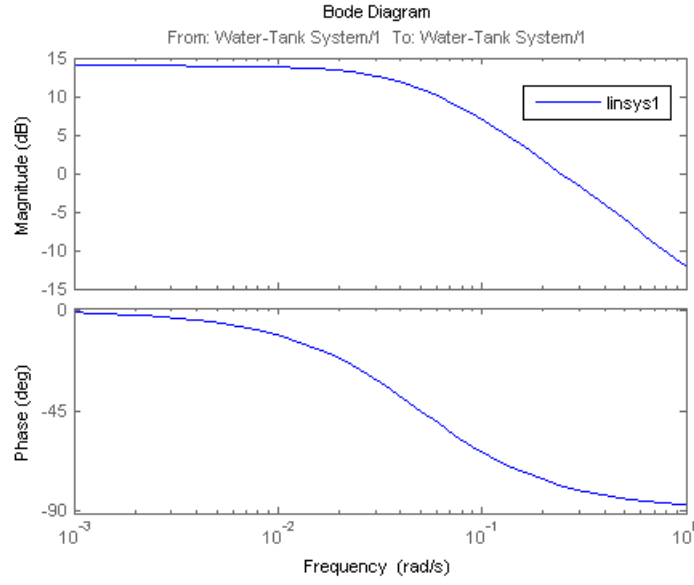
4. In the **Plot Result** drop-down menu, select **New Bode**.



5 Linearize the plant.

Click .

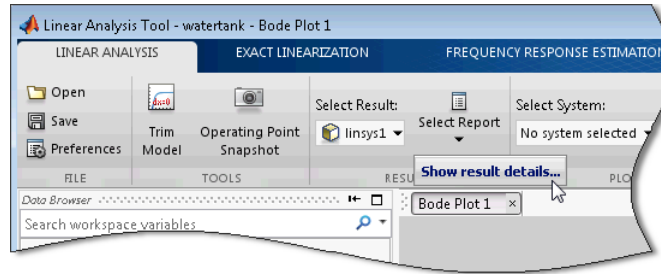
The Bode plot of the linearized plant appears.



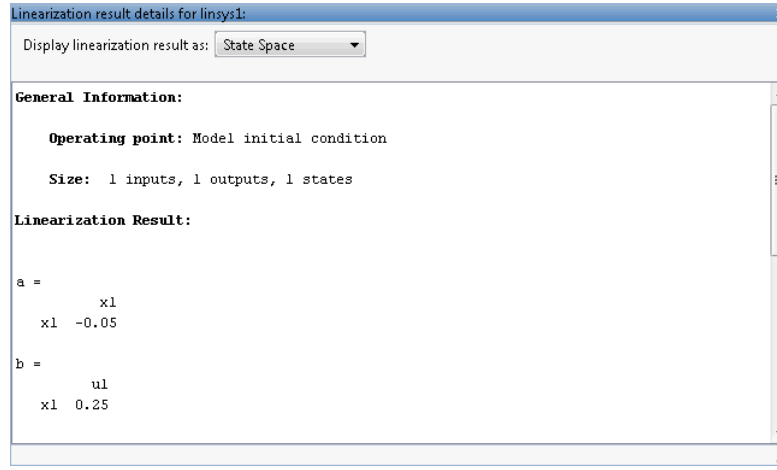
The linearization result, `linsys1`, appears in the **Linear Analysis Workspace**.

6 View the resulting linear plant model.

In the **Linear Analysis** tab, choose `linsys1` from the **Select Result** list. In the **Select Report** list, choose `Show result details`.



The Linearization results dialog box for `linsys1` opens.



Tip Drag and drop `linsys1` from the **Linear Analysis Workspace** to the **MATLAB Workspace** to export it to the base workspace for further analysis.

7 Close the Simulink model.

```
bdclose(sys);
```

Related Examples

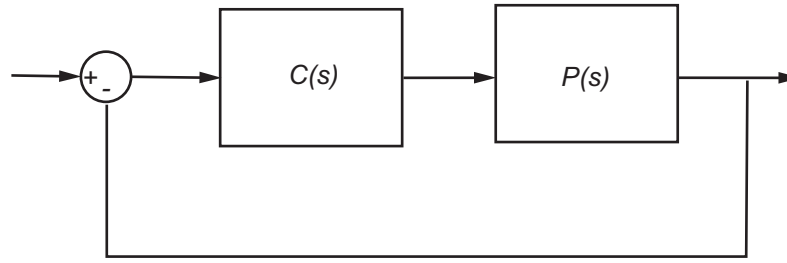
- "Linearize Simulink Model" on page 2-43
- "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

Open-Loop Response of Control System for Stability Margin Analysis

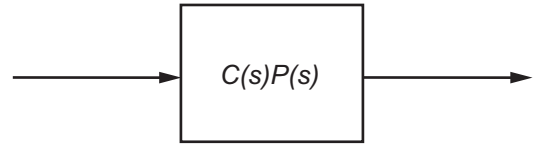
In this section...
"What Is Open-Loop Response?" on page 2-37
"Compute Open-Loop Response" on page 2-38

What Is Open-Loop Response?

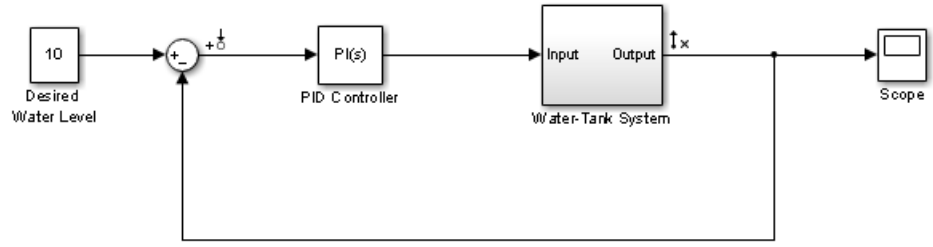
Open-loop response is the combined response of the plant and the controller, excluding the effect of the feedback loop. For example, the next block diagram shows a single-loop control system.



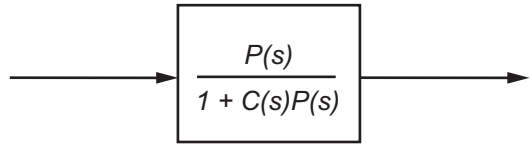
Open-loop response corresponds to the linear response of the plant and the controller. If *C(s)* and *P(s)* are linear, the corresponding linear systems is *C(s)P(s)*.



In Simulink Control Design, the linearization I/O points and the loop opening that correspond to open-loop response look something like this:



However, if there is no loop opening at the output of Water-Tank System block, the resulting linear model is different:



Compute Open-Loop Response

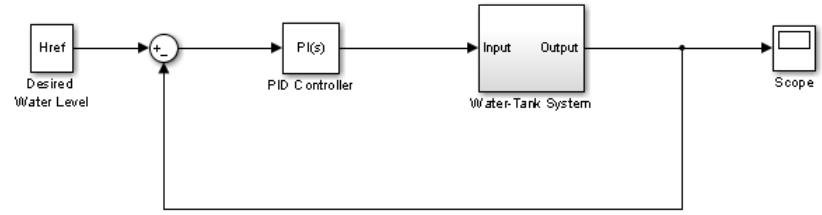
This example shows how to use the Linear Analysis Tool to analyze the open-loop response of a control system.

Compute a linear model of the combined controller-plant system without the effects of the feedback signal. Use a Bode plot of the resulting linear model to see the open-loop response.

1 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.



2 Open the Linear Analysis Tool for the model.

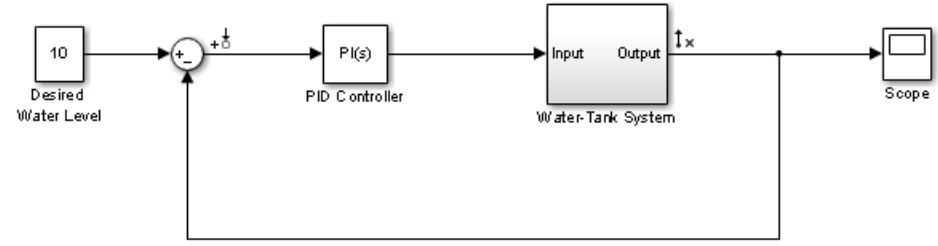
In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

3 In the Simulink Editor, define the portion of the model to linearize:

a Right-click the PID Controller block input signal (the output of the Sum block). Select **Linear Analysis Points > Input Perturbation**.

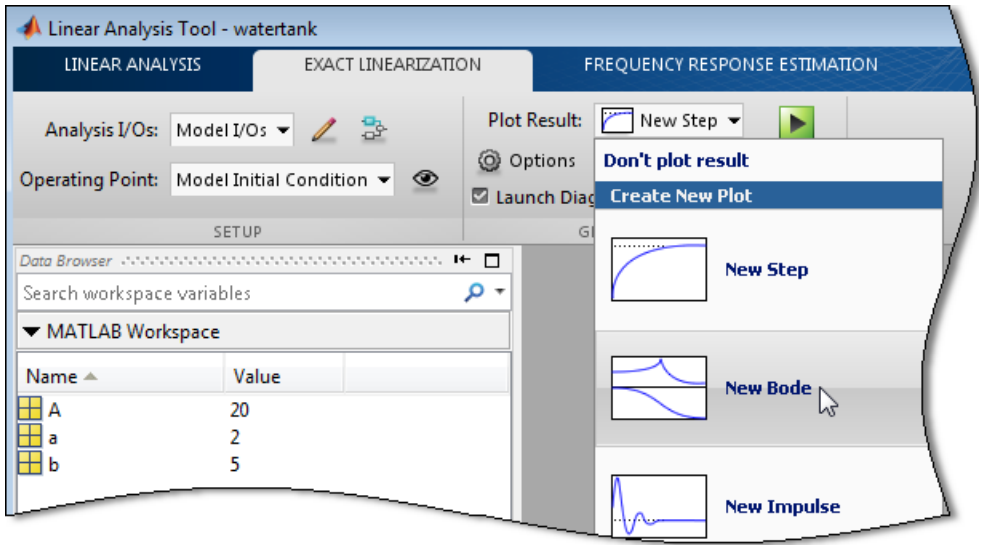
b Right-click the Water-Tank System output signal, and select **Linear Analysis Points > Open-loop Output**.

Annotations appear in the model indicating which signals are designated as linearization I/O points.



Tip Alternatively, if you do not want to introduce changes to the Simulink model, you can specify the linearization I/O points in the Linear Analysis Tool. See "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-20.

4 In the **Plot Result** list of the Linear Analysis Tool, select **New Bode**.



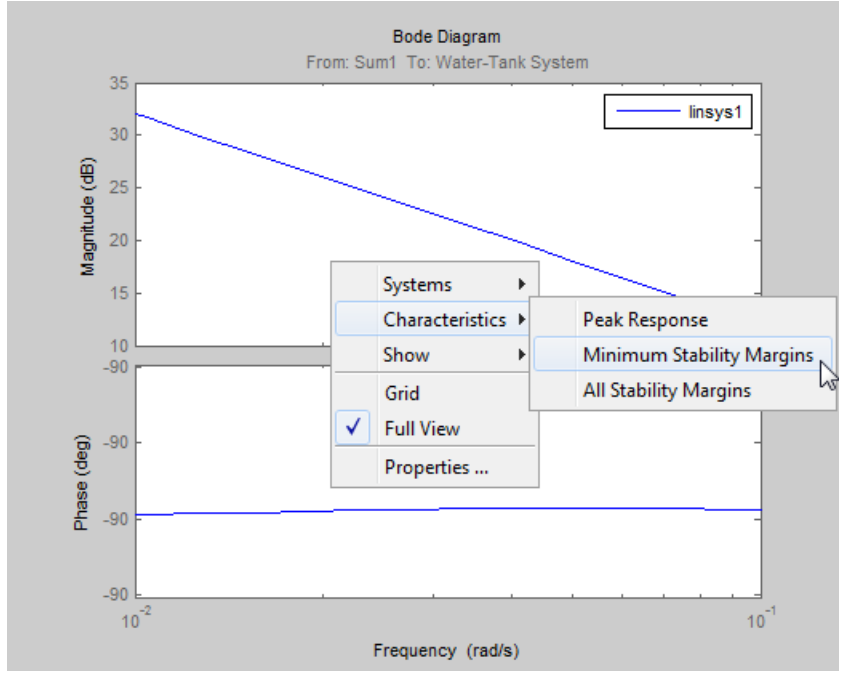
5 Linearize the model.

Click .

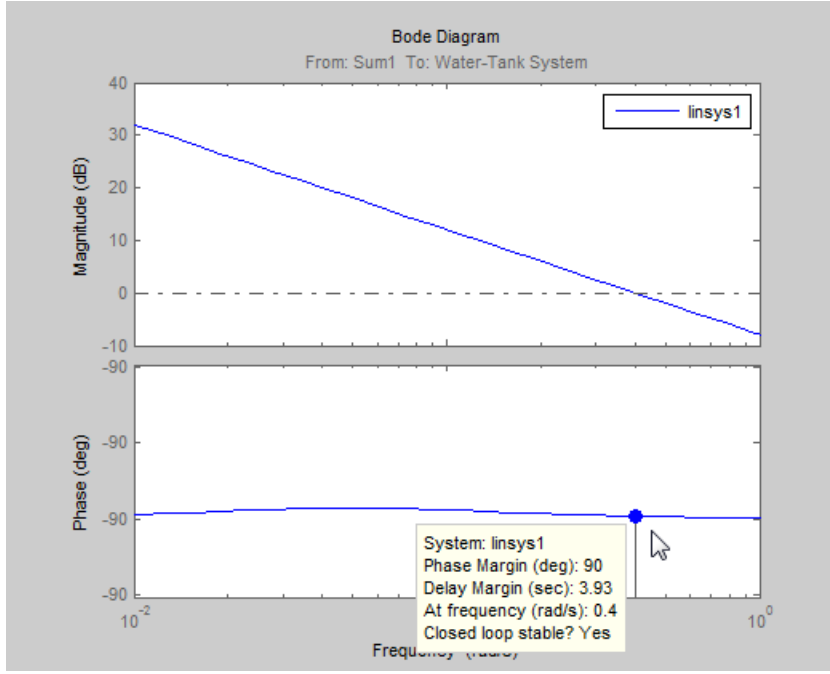
The Bode plot of the open-loop response appears.

6 View the minimum stability margins for the model.

Right-click the plot and select **Characteristics > Minimum Stability Margins**.



The Bode plot displays the phase margin marker. Click the marker to show a data tip that contains the phase margin value.



7 Close Simulink model.

```
bdclose(sys);
```

Related Examples

- “Linearize Simulink Model” on page 2-43
- “Plant Linearization” on page 2-33
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-47

Linearize at Model Operating Point

In this section...

"Linearize Simulink Model" on page 2-43

"Visualize Bode Response of Simulink Model During Simulation" on page 2-47

Linearize Simulink Model

This example shows how to use the Linear Analysis Tool to linearize a model at the operating point specified in the model. The model operating point consists of the model initial state values and input signals.

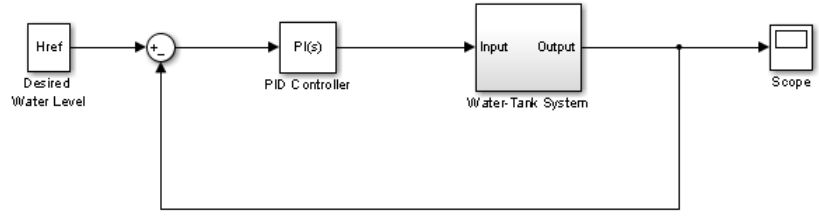
Code Alternative

Use `linearize`. For examples and additional information, see the `linearize` reference page.

1 Open Simulink model.

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

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



2 Open the Linear Analysis Tool for the model.

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

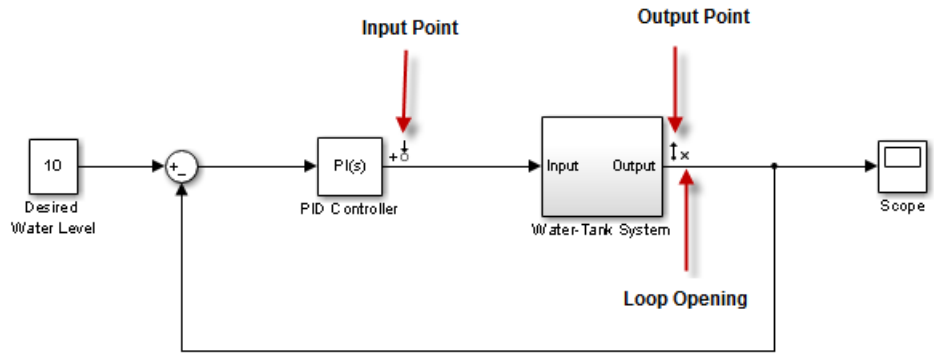
3 In the Simulink Editor, define the portion of the model to linearize:

a Right-click the PID Controller block output signal, which is the input to the plant. Select **Linear Analysis Points > Input Perturbation**.

b Right-click the Water-Tank System output signal, and select **Linear Analysis Points > Open-loop Output**.

Inserting this open loop point removes the effects of the feedback signal on the linearization without changing the model operating point.

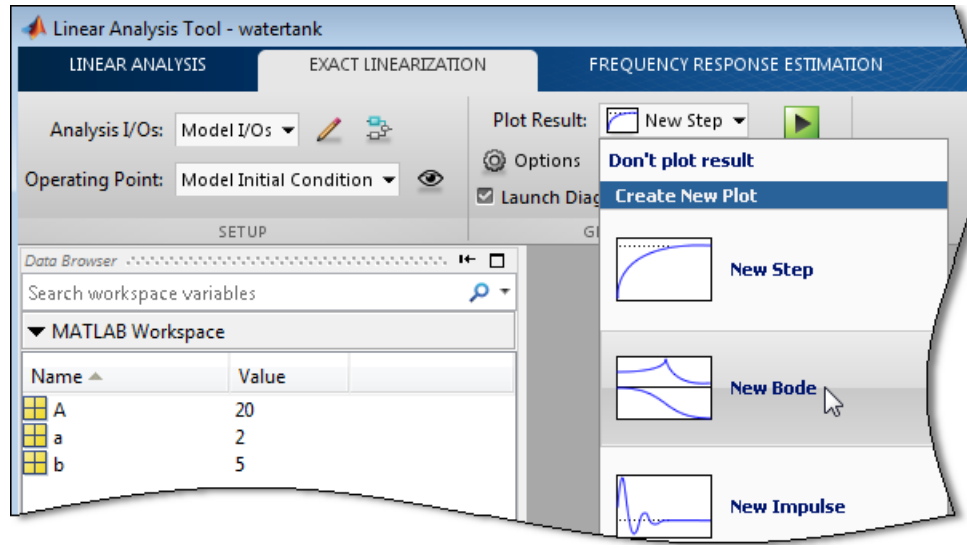
When you add linear analysis points, marker appear at their locations in the model.



Tip Alternatively, if you do not want to introduce changes to the Simulink model, you can specify the linearization I/O points in the Linear Analysis Tool. See "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-20.

4 In the **Plot Result** list, select a visualization.

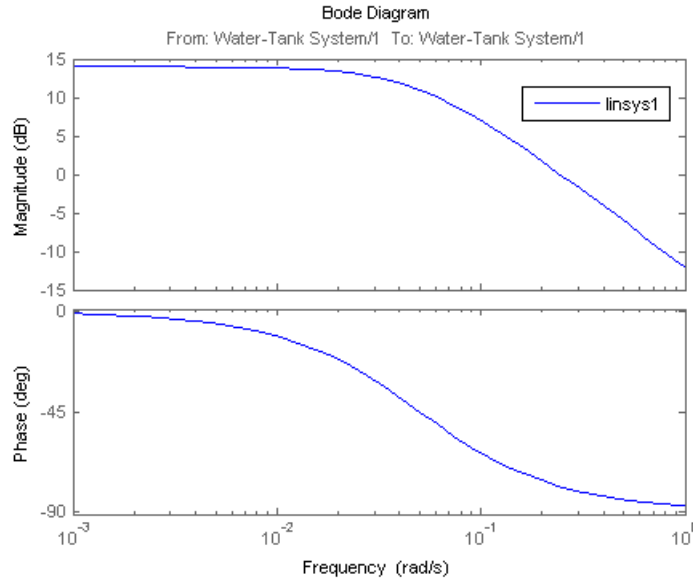
For example, select **New Bode**.



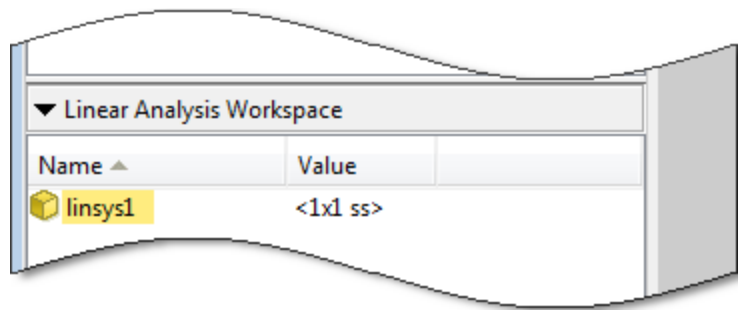
5 Linearize the model.

Click .

The Bode plot of the linearized system appears.



The linearized system, `linsys1` appears in the Linear Analysis Workspace.



6 Close Simulink model.

```
bdclose(sys);
```

Related Examples

- “Plant Linearization” on page 2-33

- "Open-Loop Response of Control System for Stability Margin Analysis" on page 2-37
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

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.

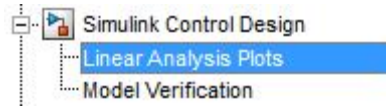
   For example:

   ```
   watertank
   ```

2. Open the Simulink Library Browser by selecting **View > Library Browser** in the model window.

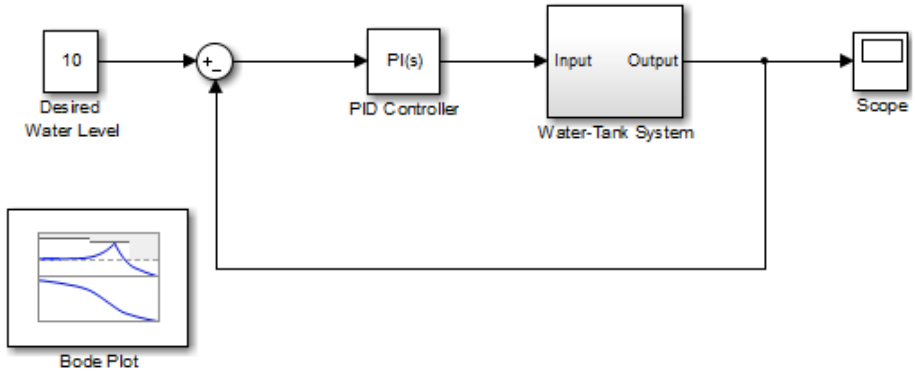
3. Add a plot block to the Simulink model.

   a. In the **Simulink Control Design** library, select **Linear Analysis Plots**.

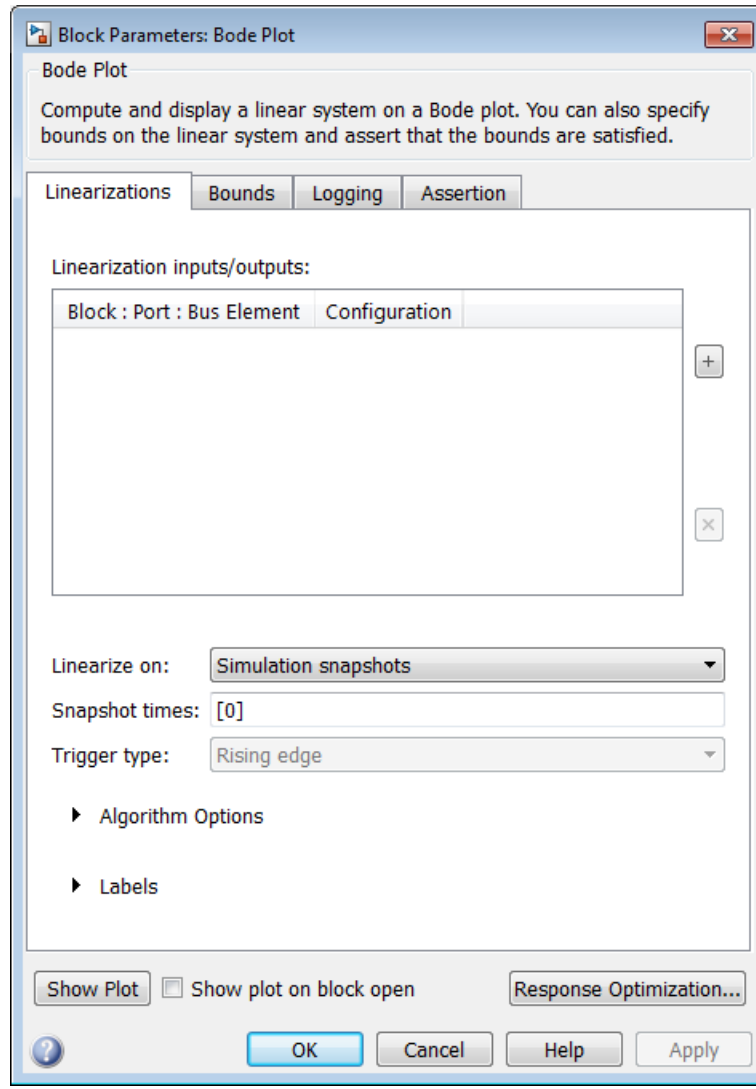
   

   b. Drag and drop a block, such as the Bode Plot block, into the model window.

   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.

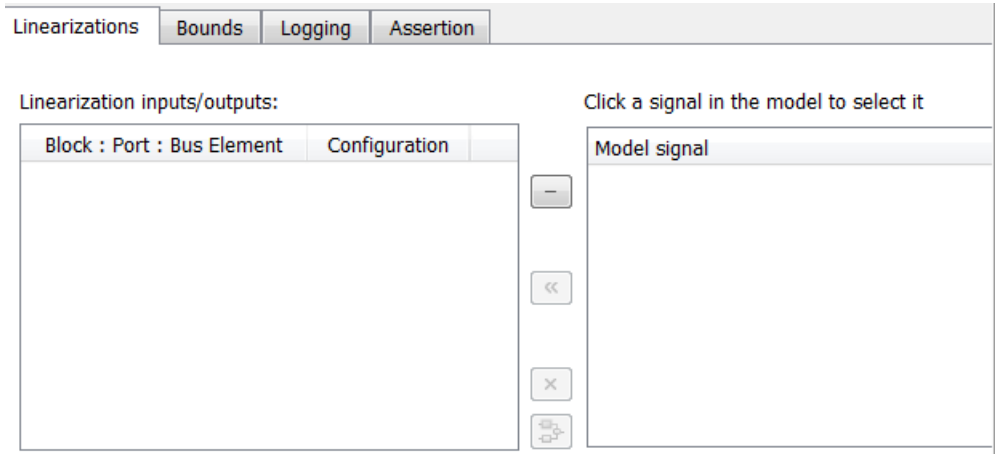
The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them.

a To specify an input:

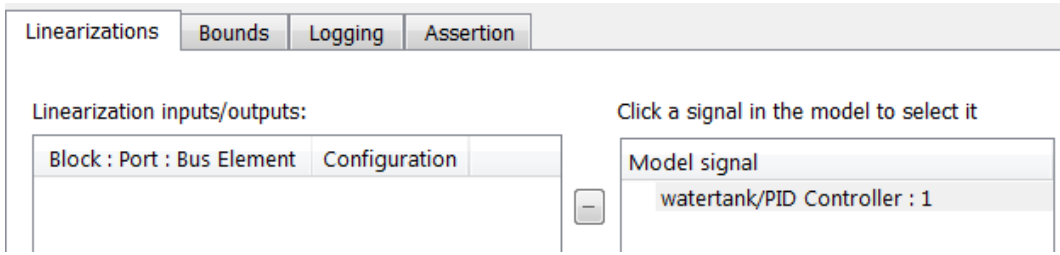
i Click adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.

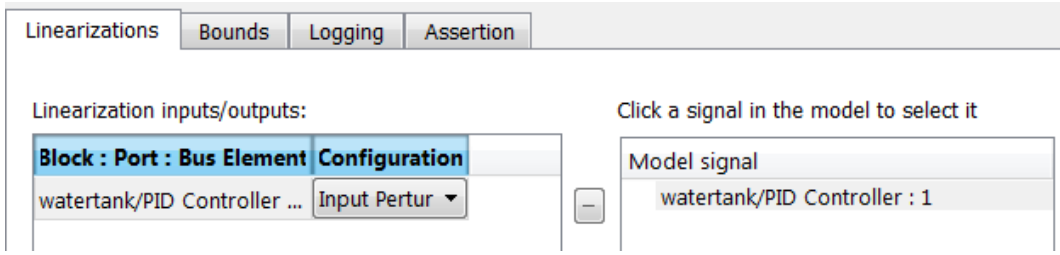


ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



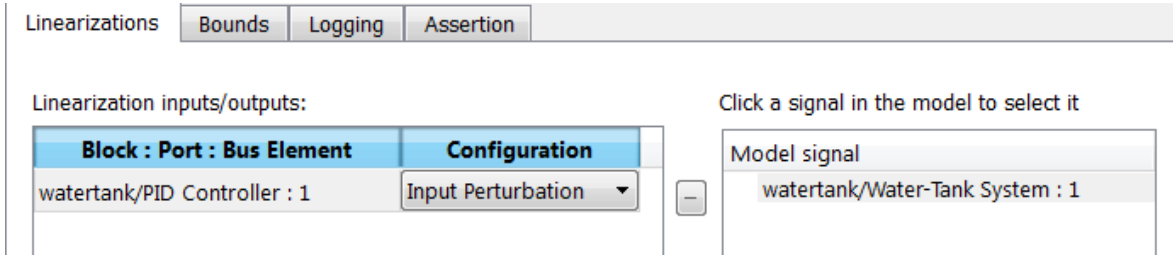
iii Click to add the signal to the **Linearization inputs/outputs** table.



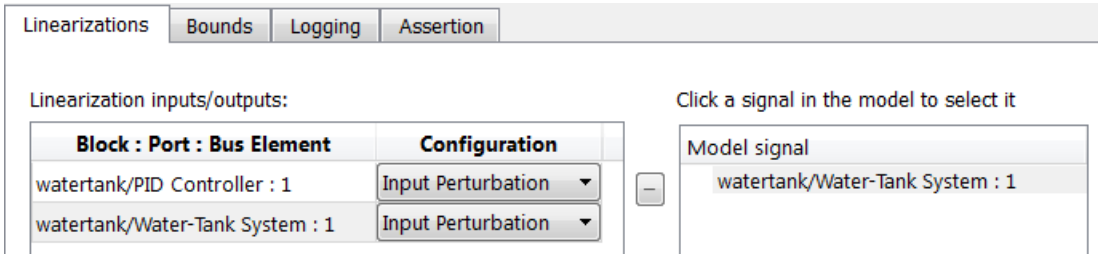
b To specify an output:

iv 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.

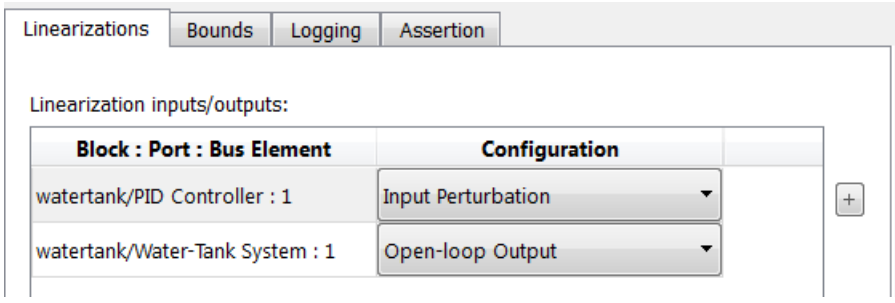


v Click to add the signal to the **Linearization inputs/outputs** table.



vi 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.

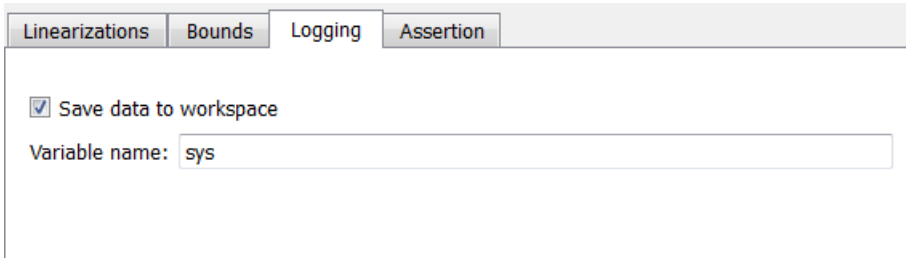
---

6 Save the linear system.

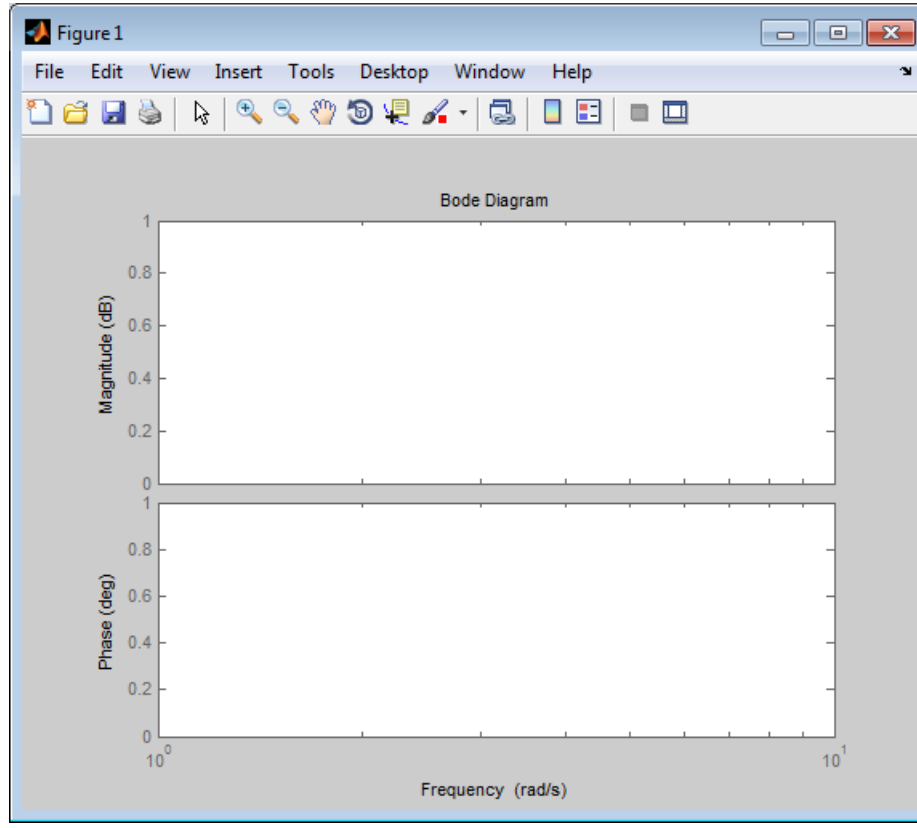
a Select the **Logging** tab.

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.

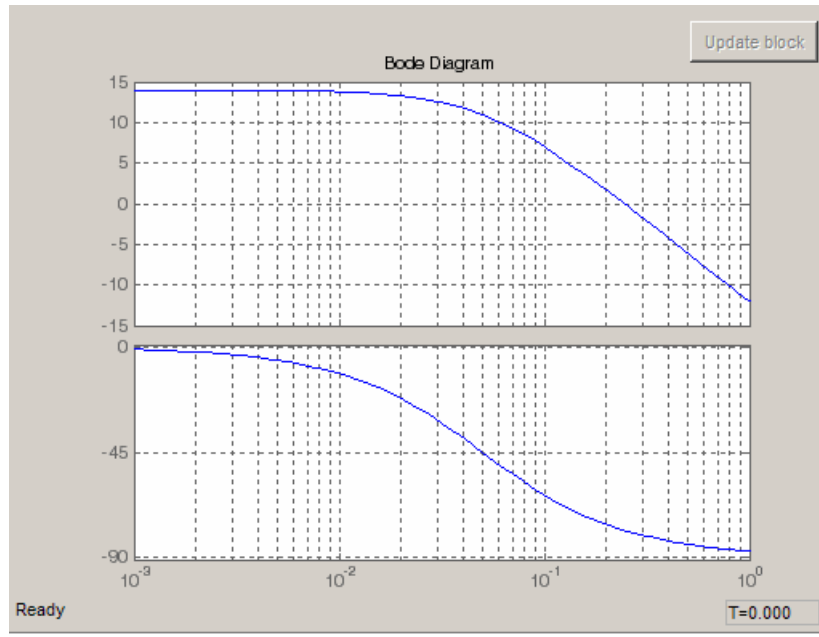


8 Plot the linear system characteristics by clicking in the plot window.

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" and the `Simulink.SimulationOutput` reference page.)

Examples and How To

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System at Trigger-Based Simulation Events" on page 2-80
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75

Linearize at Trimmed Operating Point

This example shows how to use the Linear Analysis Tool to linearize a model at a trimmed steady-state operating point (equilibrium operating point).

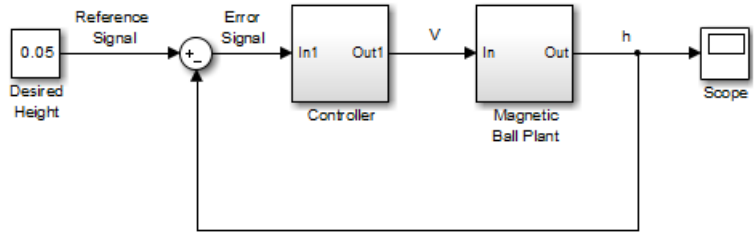
The operating point is *trimmed* by specifying constraints on the operating point values, and performing an optimization search that meets these state and input value specifications.

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**.

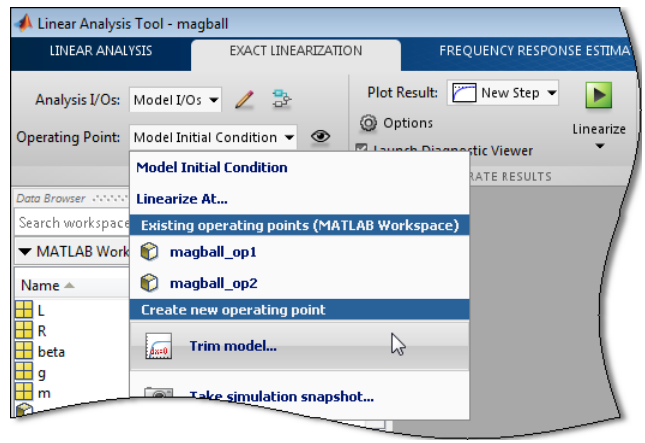
b Right-click the Magnetic Ball Plant output signal, and select **Linear Analysis Points > Open-loop Output**.

Annotations appear in the model indicating which signals are designated as linearization I/O points.

Tip Alternatively, if you do not want to introduce changes to the Simulink model, you can specify the linearization I/O points in the Linear Analysis Tool. See "Specify Portion of Model to Linearize in Linear Analysis Tool" on page 2-20.

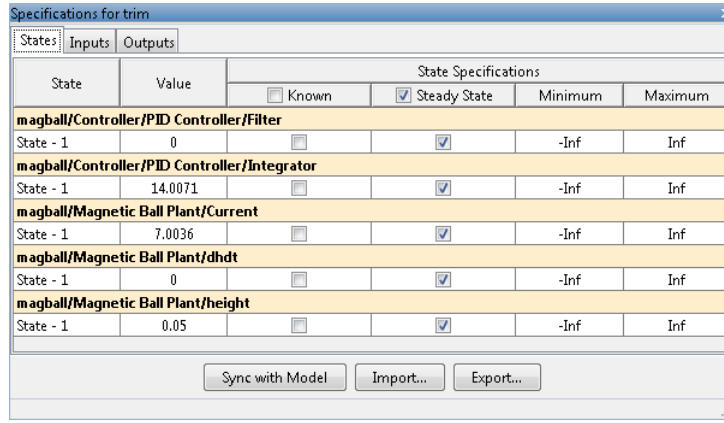
---

4 In the Linear Analysis Tool, select **Trim model**, in the **Operating Point** list.



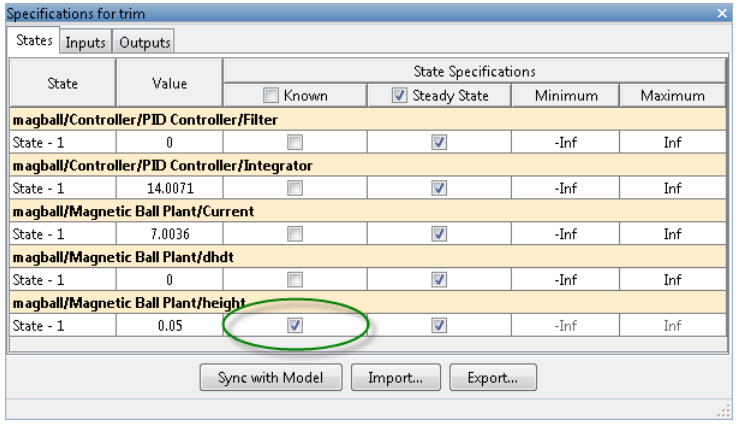
Click **Specifications**.

By default, all model states are specified to be at equilibrium (as shown in the **Steady State** column).

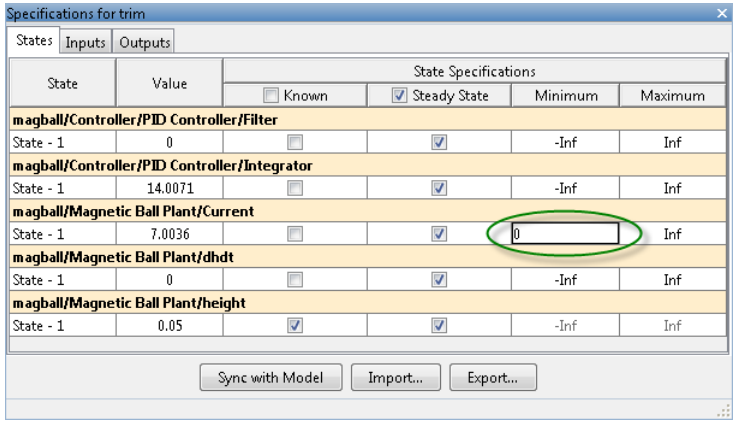


5 In the **States** tab, select **Known** for the **height** state.

The height of the ball should match the reference signal height. This height value should remain fixed during the optimization.



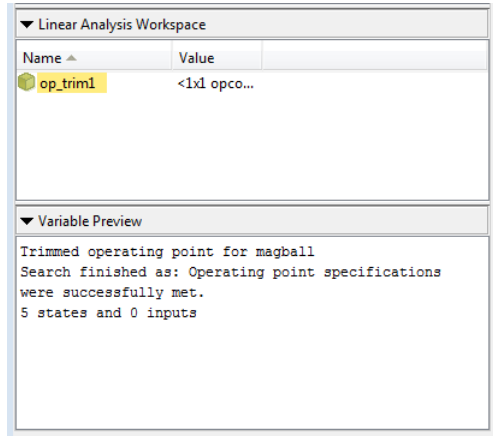
6 Enter `0` for the minimum bound of the **Current** block state.



7 Compute the operating point.

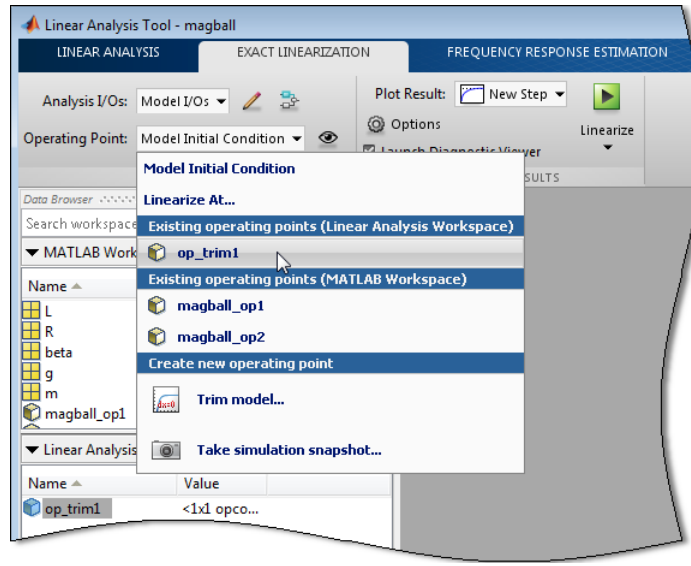
Click .

A new variable, `op_trim1`, appears in the **Linear Analysis Workspace**.



8 Select the operating point to be used for linearization.

In the **Exact Linearization** tab, select `op_trim1` from the **Operating Point** list.



9 Linearize the model at the specified operating point.

Click .

Related Examples

"Steady-State Operating Points (Trimming) from Specifications" on page 1-14

Linearize at Simulation Snapshots and Triggered Events

In this section...

"Linearize at Simulation Snapshot" on page 2-62

"Linearize at Triggered Simulation Events" on page 2-65

"Visualize Linear System at Multiple Simulation Snapshots" on page 2-69

"Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75

"Visualize Linear System at Trigger-Based Simulation Events" on page 2-80

Linearize at Simulation Snapshot

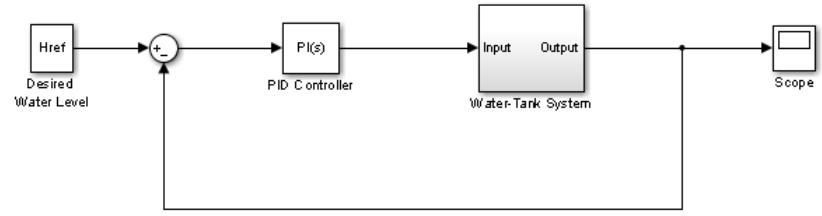
This example shows how to use the Linear Analysis Tool to linearize a model by simulating the model and extracting the state and input levels of the system at specified simulation times.

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.

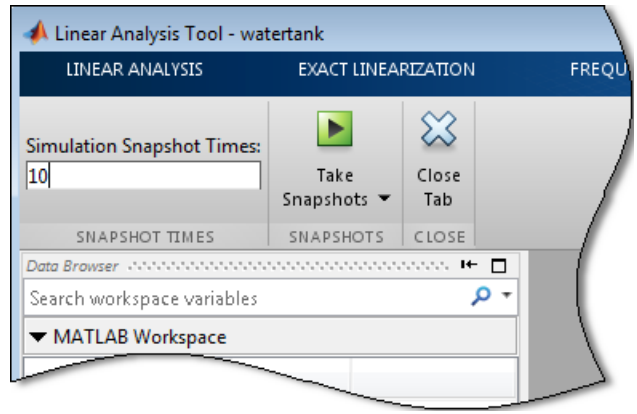
In the Simulink model window, select **Analysis > Control Design > Linear Analysis**.

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 In the Linear Analysis Tool, select **Take simulation snapshot** in the **Operating Point** list.

5 In the **Simulation Snapshot Times** field, enter `10` to extract the operating point at this simulation time.



6 Take a snapshot of the system at the specified time.

Click .

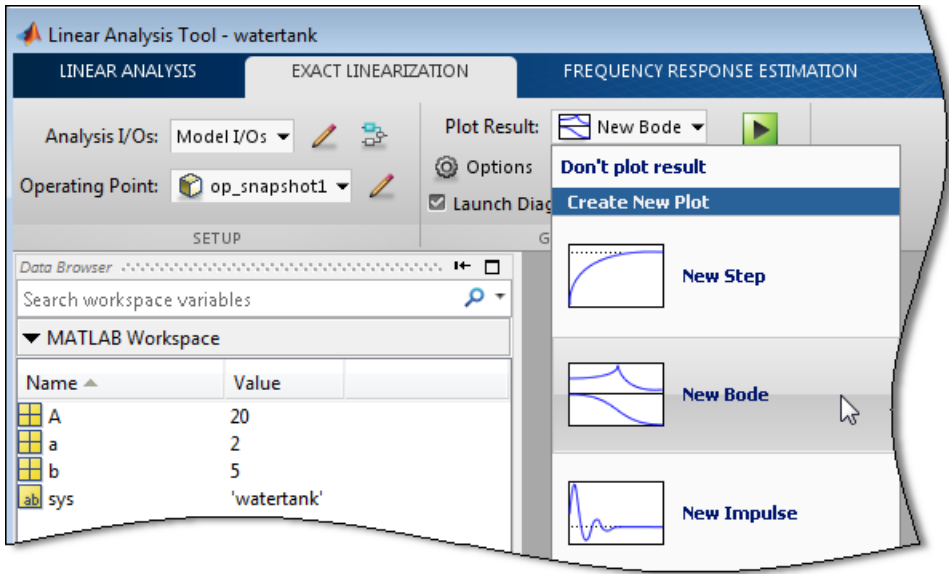
The operating point `op_snapshot1` appears in the **Linear Analysis Workspace**.

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 two linear models, one linearized at `t = 1` and the other at `t = 10`.

7 In the **Exact Linearization** tab, select `op_snapshot1` in the **Operating Point** list.

8 In the **Plot Result** list, select a visualization.

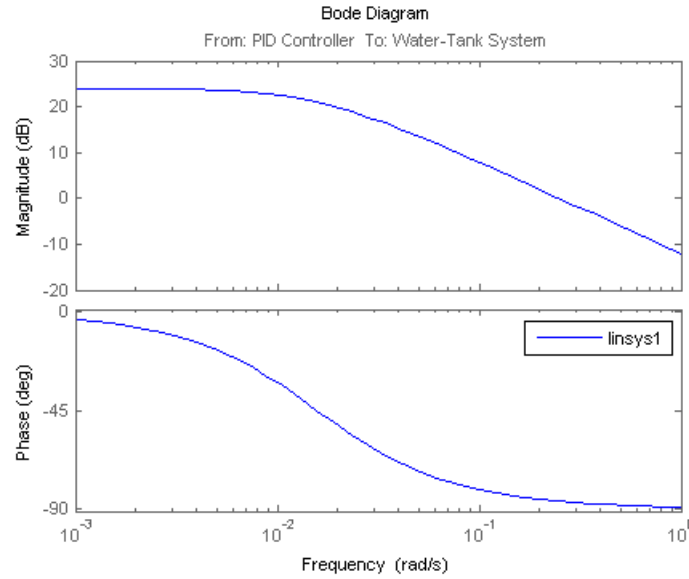
Select **New Bode**.



9 Linearize the model.

Click .

The Bode plot of the linearized system appears. This Bode plot looks like a stable first-order response, as expected.



10 Double click `linsys1` in the **Linear Analysis Workspace** to see the state space representation of the linear model.

11 Close Simulink model.

```
bdclose(sys);
```

Related Examples

- "Linearize at Triggered Simulation Events" on page 2-65
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- "Visualize Linear System at Trigger-Based Simulation Events" on page 2-80

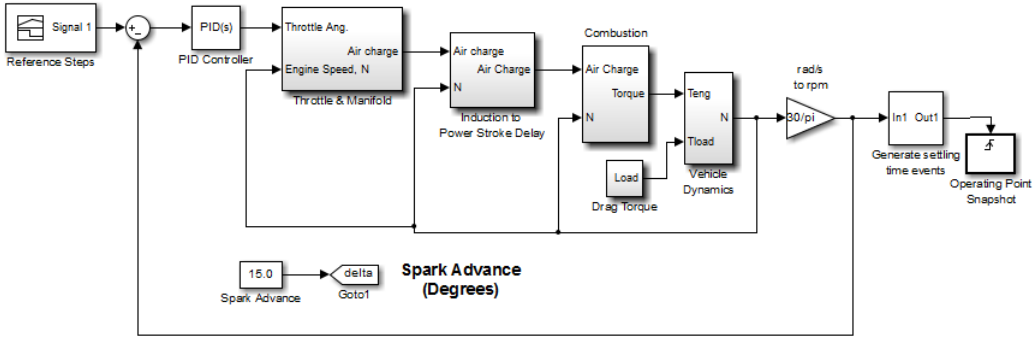
Linearize at Triggered Simulation Events

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

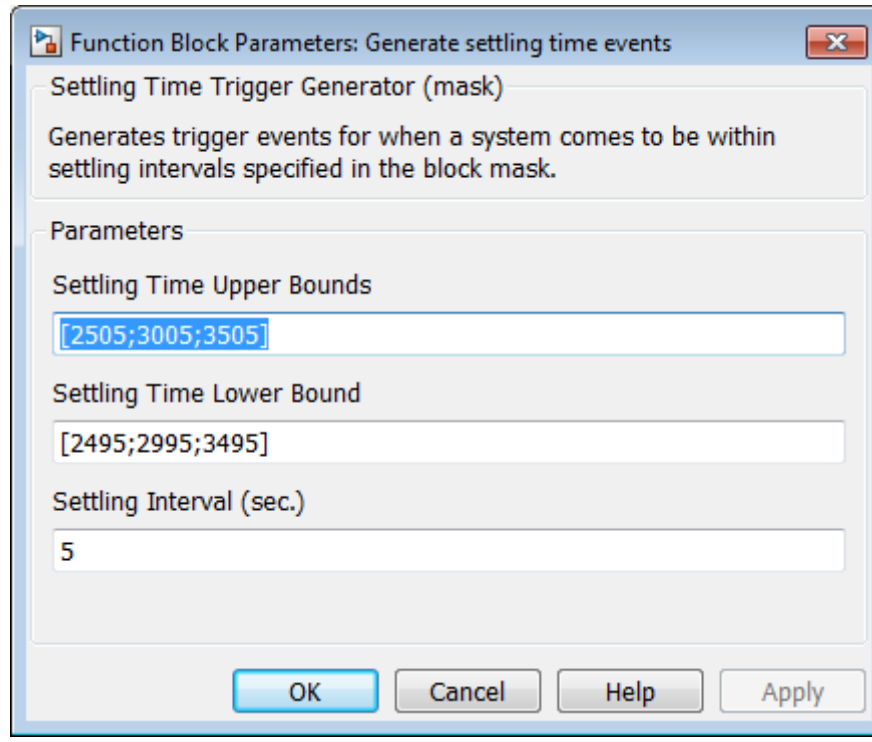
or function-call events. Specifically, the model will be linearized at the steady-state operating points 2500, 3000, and 3500 rpm.

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 5 seconds.



The model already includes the 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.

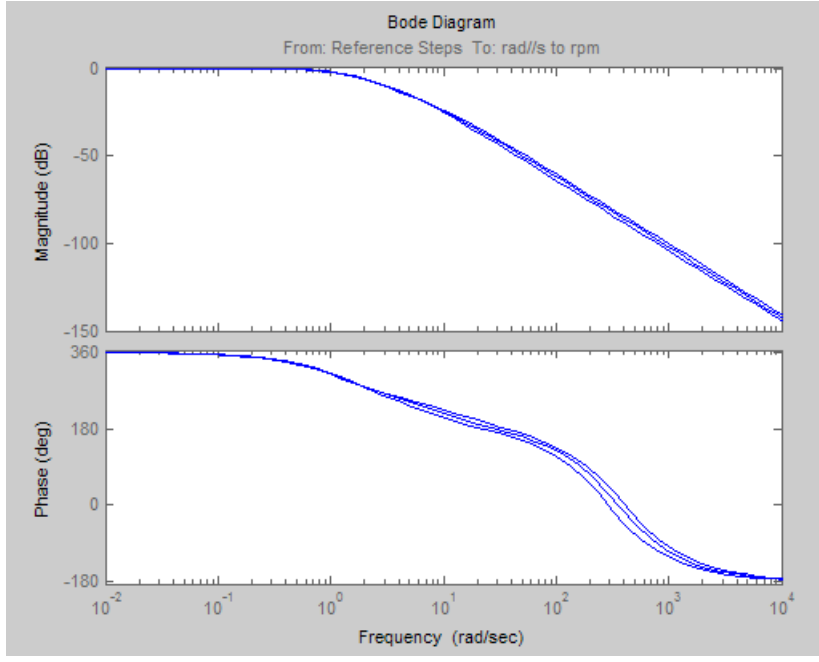
```
io(1) = linio('scdspeedtrigger/Reference Steps',1,'input');
io(2) = linio('scdspeedtrigger/rad//s to rpm',1,'output');
```

4 Linearize the model.

```
linsys = linearize(sys,op(1:3),io);
```

5 Compare linearized models at 500, 3000, and 3500 rpm using Bode plots of the closed-loop transfer functions.

```
bode(linsys);
```



Related Examples

- "Linearize at Simulation Snapshot" on page 2-62
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- "Visualize Linear System at Trigger-Based Simulation Events" on page 2-80

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
   ```

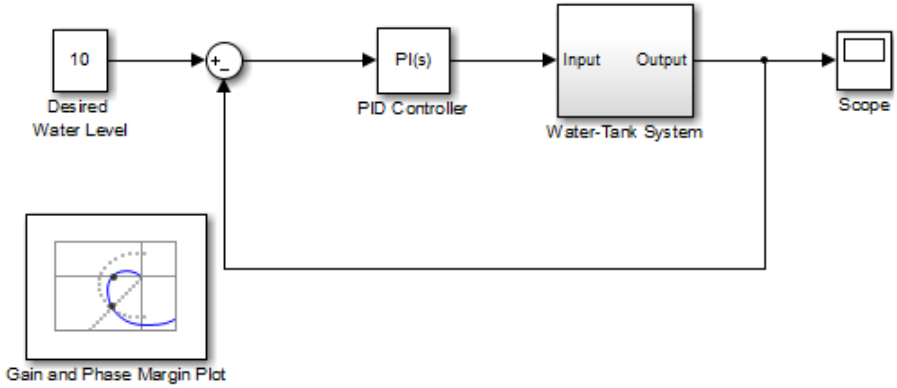
2. Open the Simulink Library Browser by selecting **View > Library Browser** in the model window.

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.

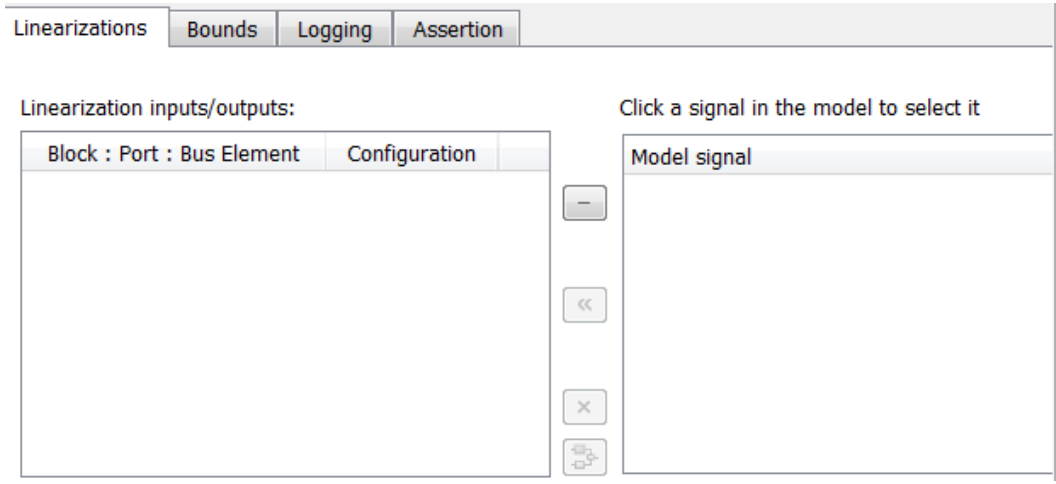
The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them.

a To specify an input:

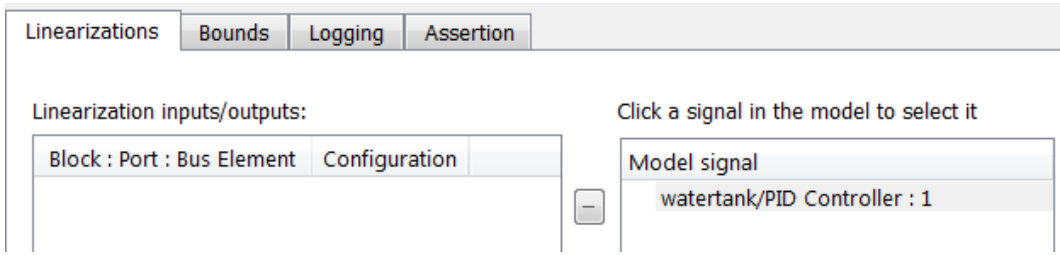
i Click adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.

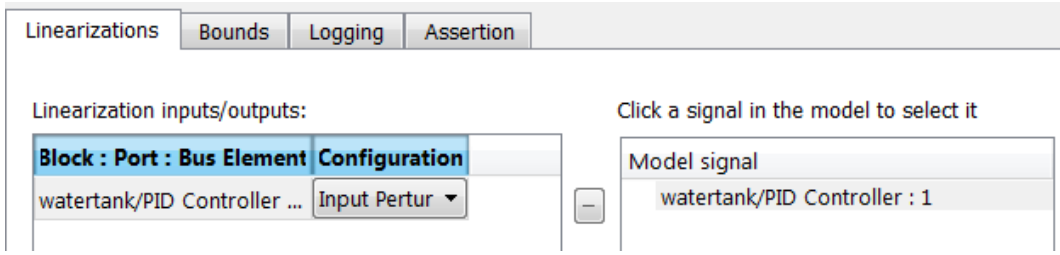


ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



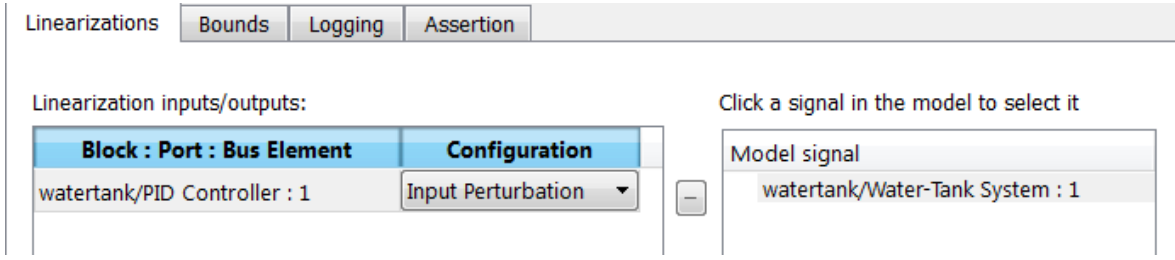
iii Click to add the signal to the **Linearization inputs/outputs** table.



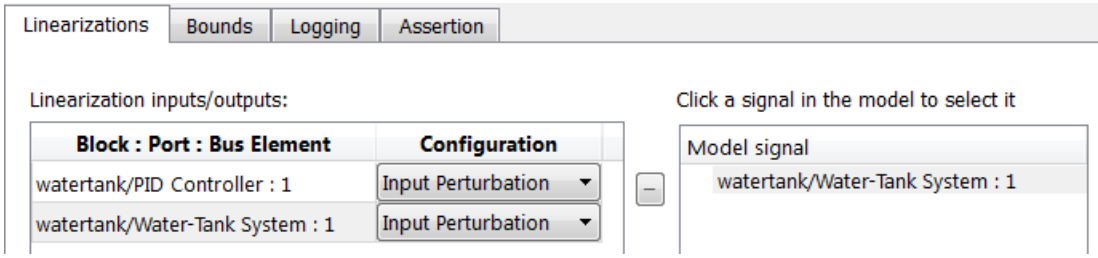
b To specify an output:

iv 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.

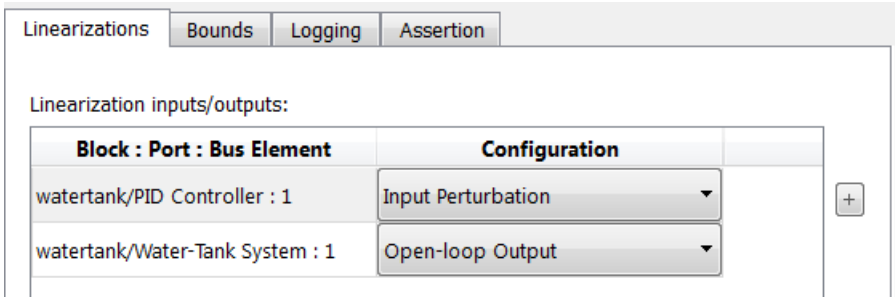


v Click to add the signal to the **Linearization inputs/outputs** table.



vi 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.

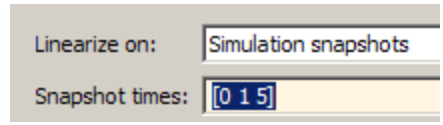
Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear in the **Linearization inputs/outputs** table.

---

6 Specify simulation snapshot times.

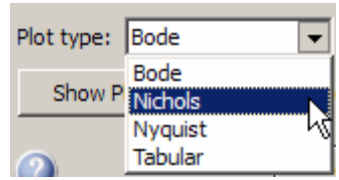
a In the **Linearizations** tab, verify that `Simulation snapshots` is selected in **Linearize on**.

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**



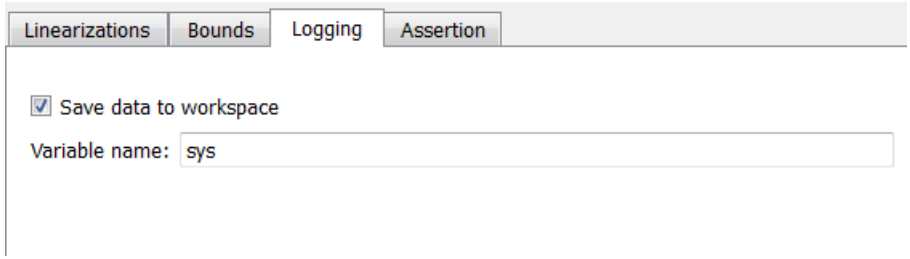
b Click **Show Plot** to open an empty Nichols plot.

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.

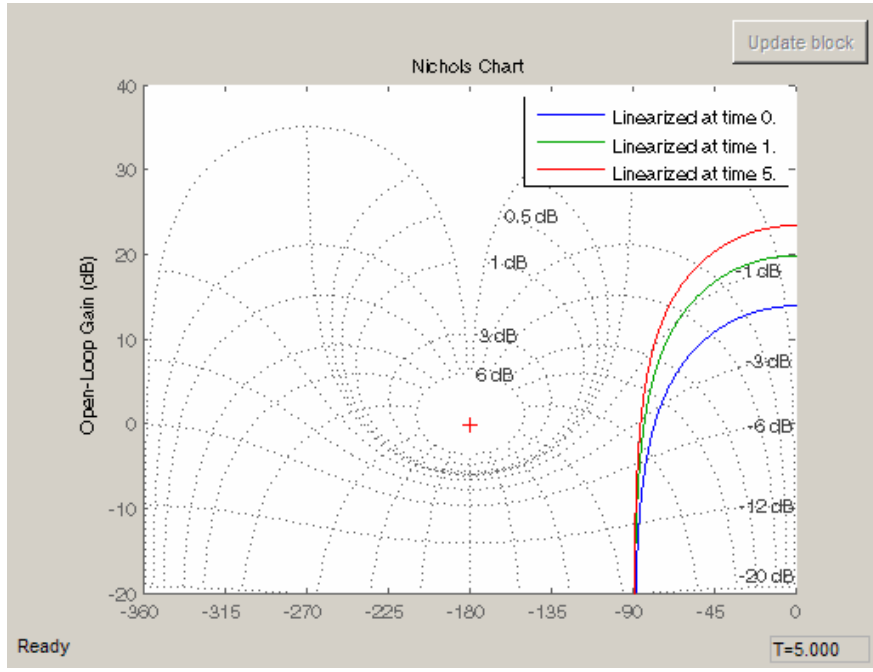
The **Logging** tab now resembles the following figure.



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 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" and the `Simulink.SimulationOutput` reference page.)

Examples and How To

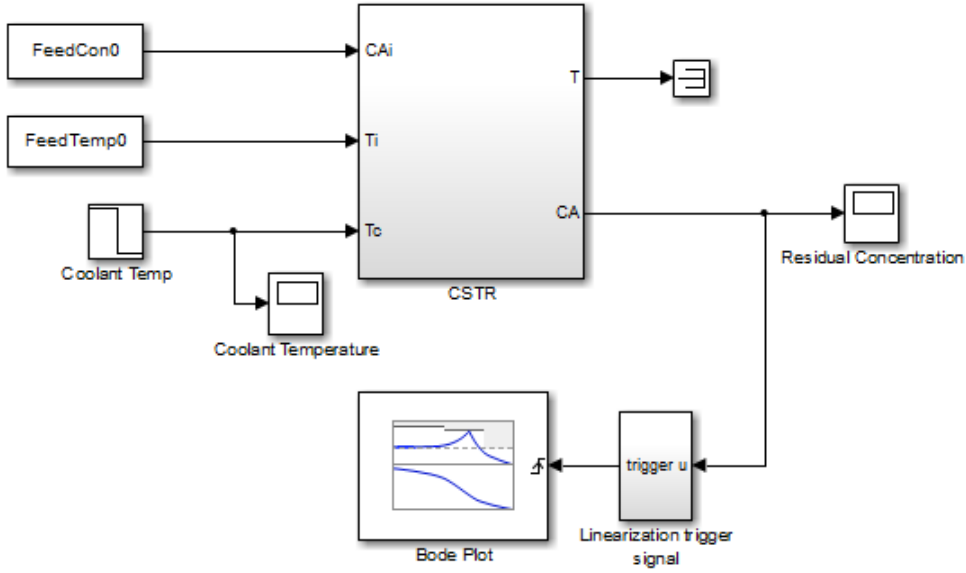
- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47
- "Visualize Linear System at Trigger-Based Simulation Events" on page 2-80
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- "Linearize at Simulation Snapshot" on page 2-62
- "Linearize at Triggered Simulation Events" on page 2-65

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:

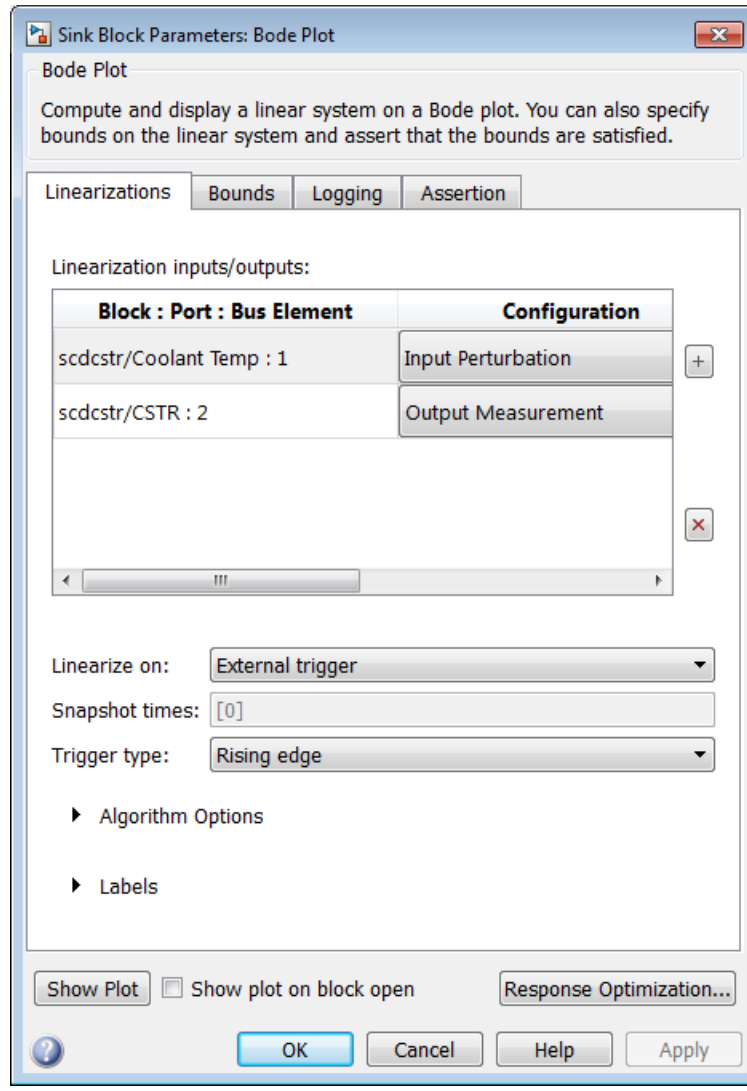
```
scdcstr
```



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.

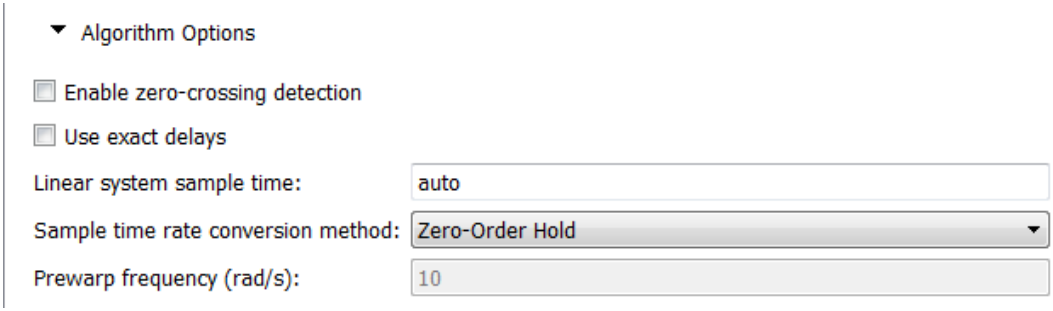


To learn more about the block parameters, see the block reference pages.

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.

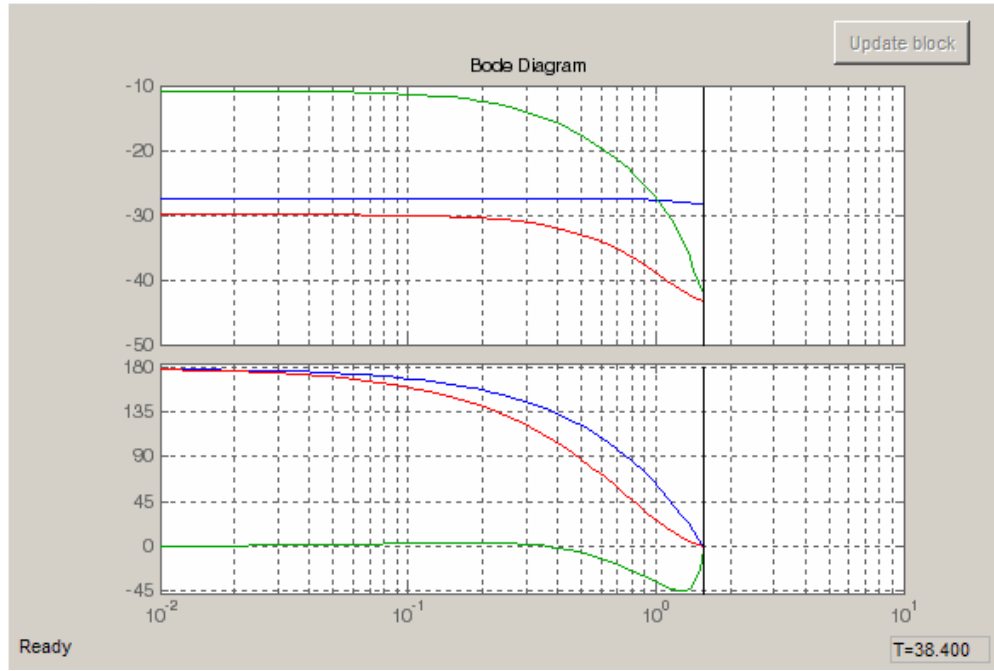
3 Click **Show Plot** to open an empty Bode plot window.

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.

Examples and How To

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System at Trigger-Based Simulation Events" on page 2-80
- "Linearize at Simulation Snapshot" on page 2-62
- "Linearize at Triggered Simulation Events" on page 2-65

Visualize Linear System at Trigger-Based Simulation Events

Plotting Linear System Characteristics of a Chemical Reactor shows how to plot the Bode magnitude and phase of a reactor. The reactor transitions through different operating points corresponding to trigger-based simulation events.

Examples and How To

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- "Linearize at Simulation Snapshot" on page 2-62
- "Linearize at Triggered Simulation Events" on page 2-65

Ordering States in Linearized Model

In this section...

"Control State Order of Linearized Model using Linear Analysis Tool" on page 2-81

"Control State Order of Linearized Model using MATLAB Code" on page 2-84

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.

```
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);
```

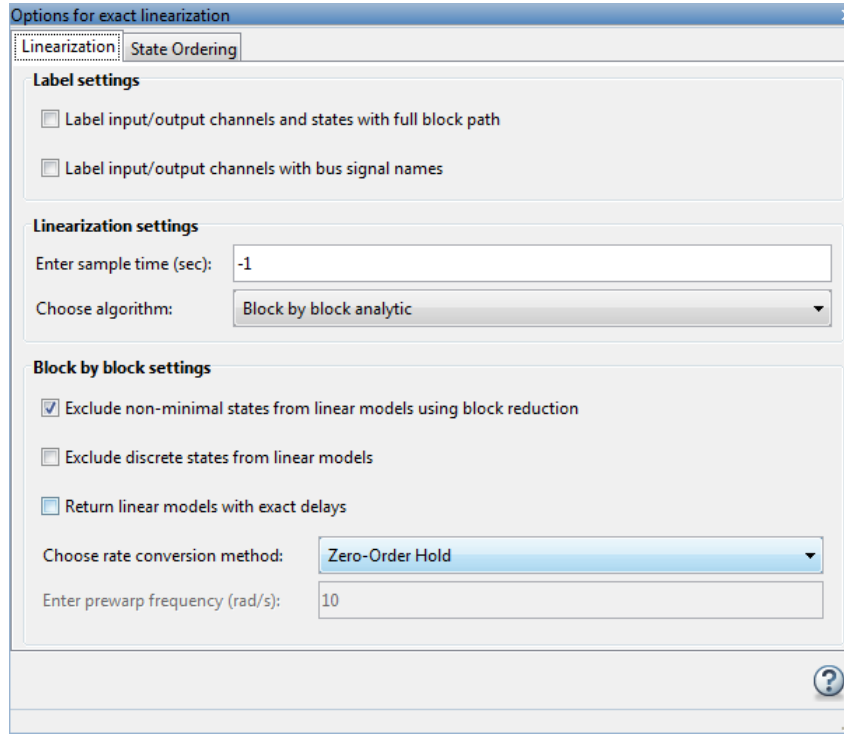
These commands specify the plant linearization and compute the steady-state operating point.

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.

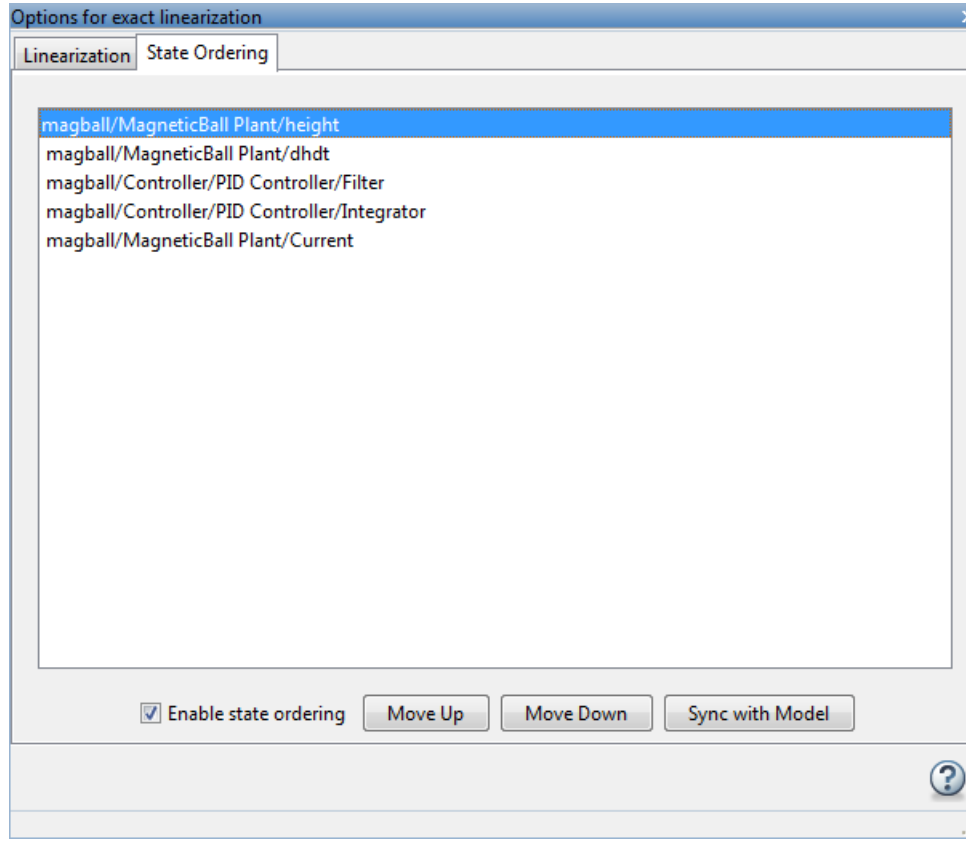
In the **Exact Linearization** tab, click **Options**.



4 In the **State Ordering** tab, select the **Enable state ordering** check box.

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.



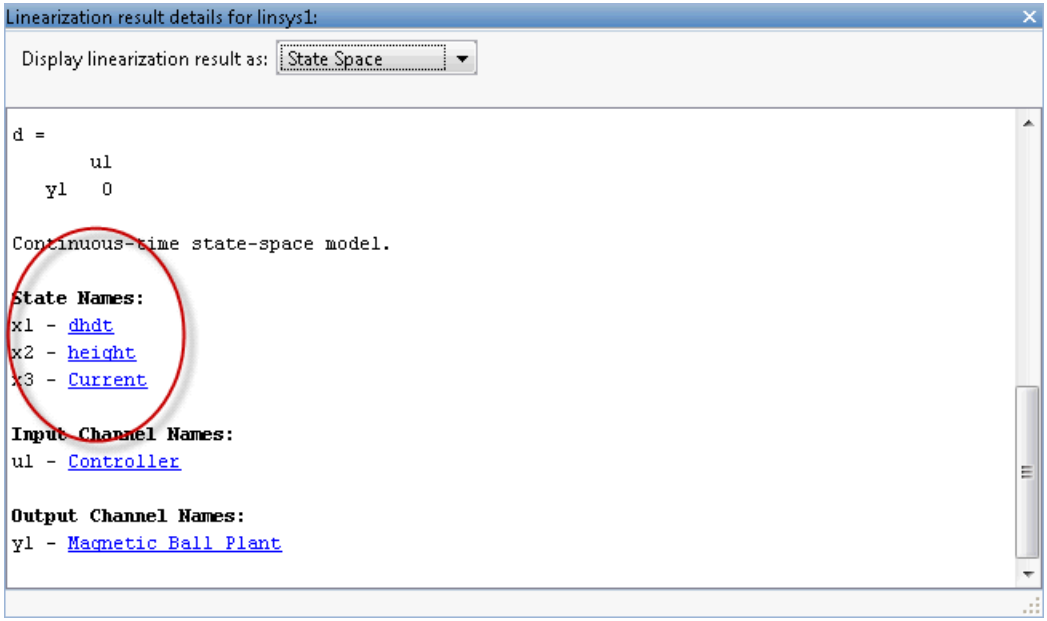
6 Linearize the model.

In the Linear Analysis Tool, click .

A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

7 View the order of the model states for the linearized model by opening the Linearization result dialog box.

Click the **Linear Analysis** tab. Choose **Show result details** in the **Select Report** list.



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'
```

Time-Domain Validation of Linearization

In this section...

"Validate Linearization in Time Domain" on page 2-86

"Choosing Time-Domain Validation Input Signal" on page 2-88

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);
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.

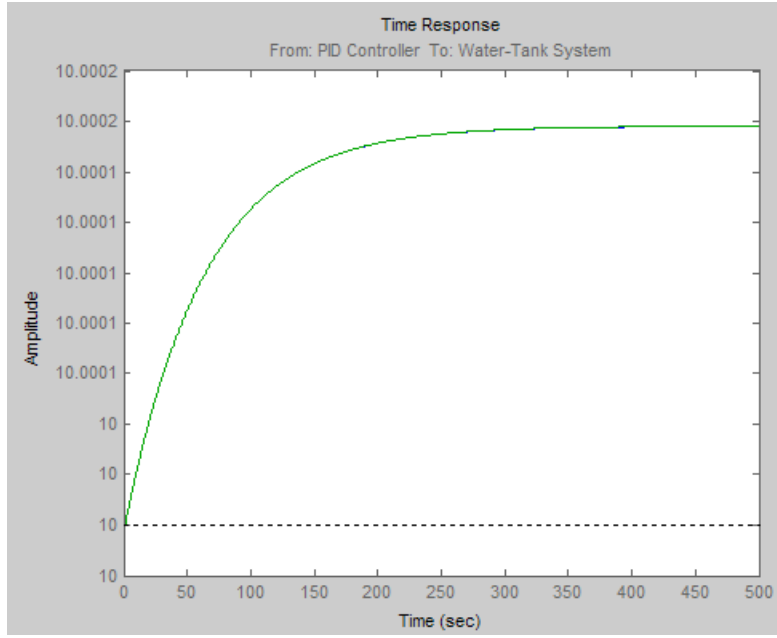
```
[~,simout] = frestimate(sys,op,sys_io,input);
```

`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)
```

The step response of the nonlinear model and linearized model are close, which validates that the linearization is accurate.



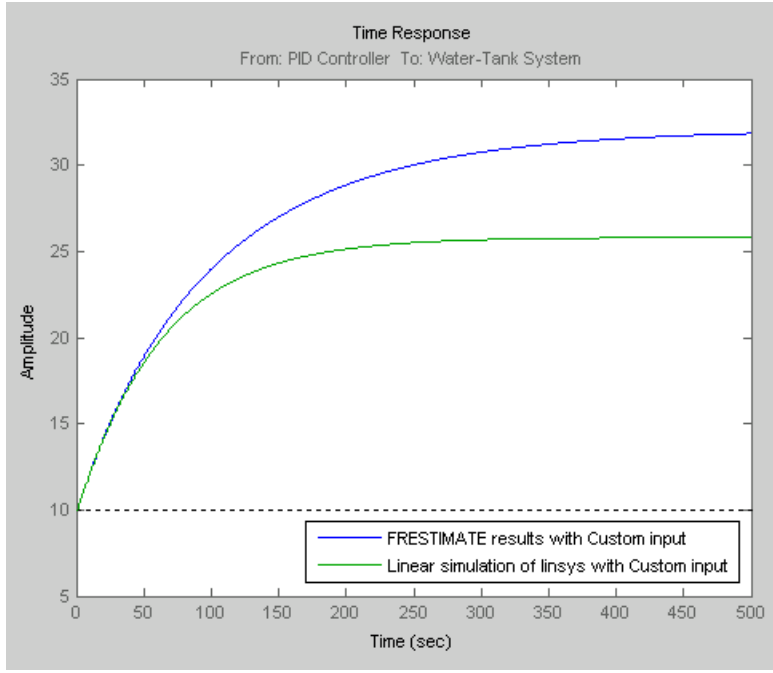
5 Increase the amplitude of the step signal from `1.0e-005` to `1`.

```
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.

This result indicates that linear model obtained using exact linearization does not behave linearly you begin 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.

Frequency-Domain Validation of Linearization

In this section...

"Validate Linearization in Frequency Domain using Linear Analysis Tool" on page 2-90

"Choosing Frequency-Domain Validation Input Signal" on page 2-94

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 model initial conditions. 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.

Step 1. Linearize Simulink model.

- Open a Simulink model.

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

- Open the Linear Analysis Tool for the model.

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

- Select a visualization for the linearized model.

  In the **Plot Result** list, choose **New Bode**.

- Linearize the model.

  Click .

  A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

The software used the model initial conditions as the operating point to generate `linsys1`.

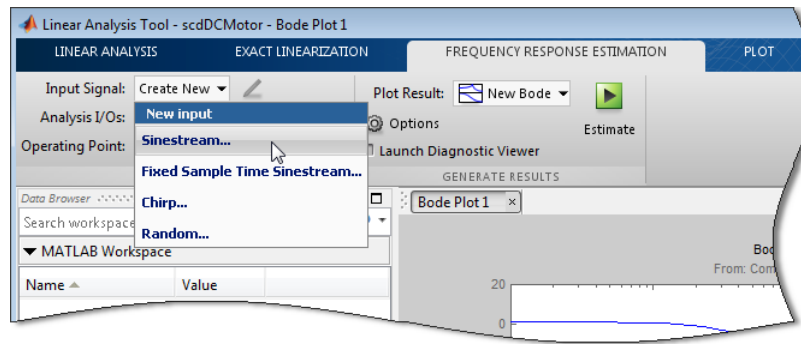
Step 2. Create sinestream input signal.

- Click the **Frequency Response Estimation** tab.

  In this tab, you estimate the frequency response of the model.

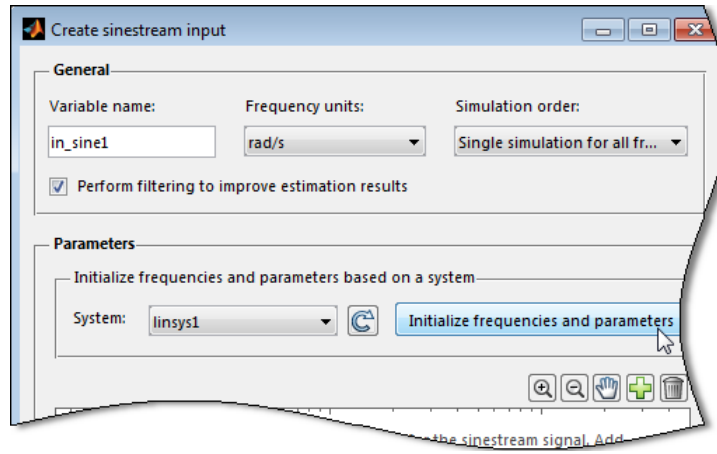
- Open the Create sinestream input dialog box.

  Select **Sinestream** from the **Input Signal** list.



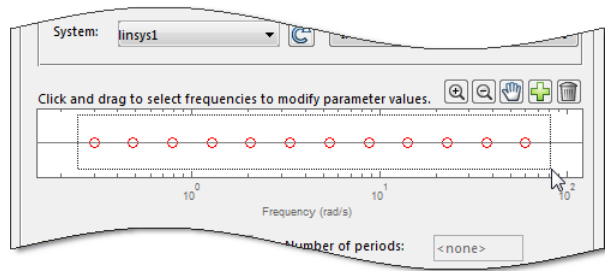
- Initialize the input signal frequencies and parameters based on the linearized model.

  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`.

- In the Frequency content viewer of the Create sinestream input dialog box, select all the frequency points.



- Modify the amplitude of the input signal.

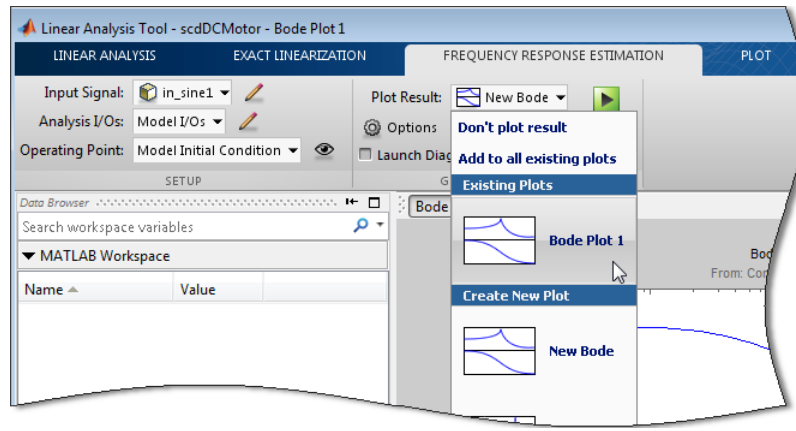
  Enter 1 in the **Amplitude** box.

- Click **OK**.

  The input signal `in_sine1` appears in the **Linear Analysis Workspace**.

Step 3. Select the plot to display the estimation result.

In the **Plot Result** list, choose **Bode Plot 1** to add the next computed linear system to **Bode Plot 1**.



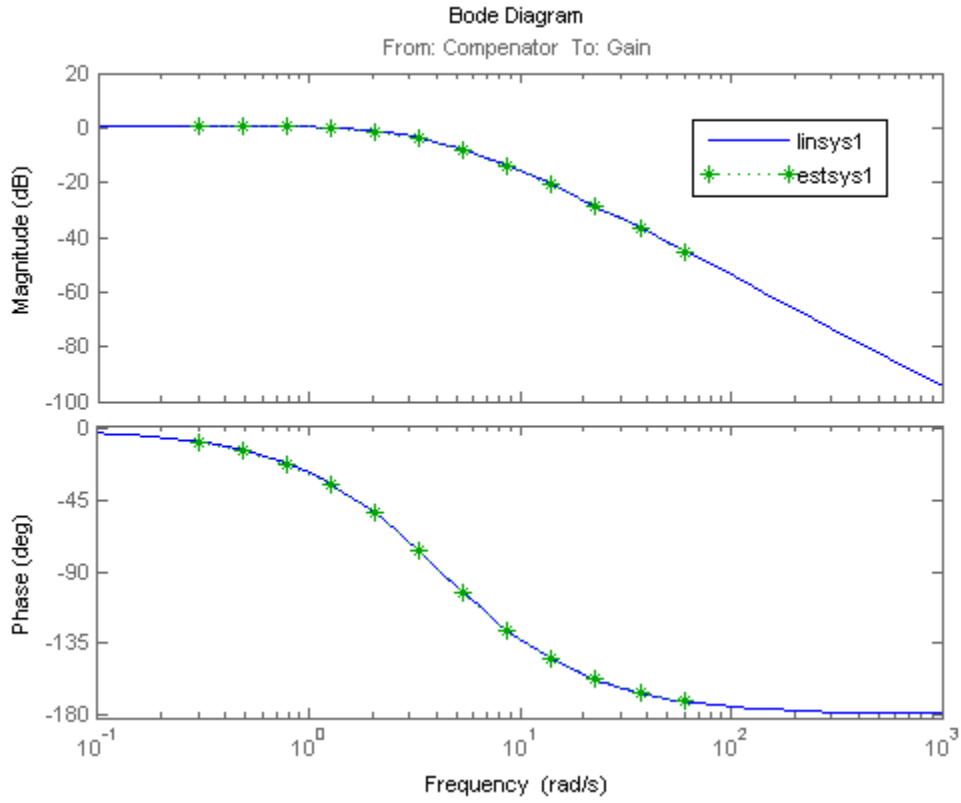
Step 4. Estimate frequency response.

Click .

The estimated system, `estsys1`, appears in the **Linear Analysis Workspace**.

Step 5. 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.

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.

Concepts

- "Creating Input Signals for Estimation" on page 3-9

Visualize Models

In this section...
"Customize Characteristics of Plot in Linear Analysis Tool" on page 2-96
"Print Plot to MATLAB Figure in Linear Analysis Tool" on page 2-96
"Generate Additional Response Plots of Linearized System" on page 2-97
"Add Linear System to Existing Response Plot" on page 2-98

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:

1. In the Linear Analysis Tool, in the **Figures** tab, in the select the plot you want to customize.
2. In the **Properties** section, click **Current plot properties** to open the Property Editor.
3. 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

You can export a plot from the Linear Analysis Tool to a MATLAB figure window. To do so:

1. After creating a plot, in the **Figures** tab, select the plot you want to export.
2. In the **Print** section, print click **Print to Figure**.

A MATLAB figure window opens containing the plot.

Generate Additional Response Plots of Linearized System

This example shows how to generate additional response plots of a linearized system in the Linear Analysis Tool.

1. Linearize a model and create a plot of the response, such as shown in "Linearize at Model Operating Point" on page 2-43.

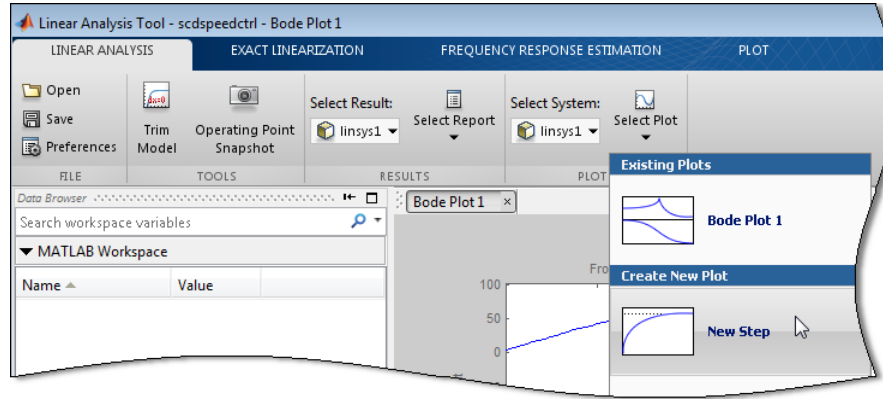
2. In the Linear Analysis Tool, click the **Linear Analysis** tab.

3. In the **Select System** list, select the system for which you want to create a new plot.

   For example, select `linsys1`.

4. In the **Select Plot** list, select a plot to add.

   For example, select **New Step**.



This action generates **Step Plot 1**, which shows the step response for `linsys1`.

Tip To view both plots at the same time, select a layout on the **View** tab, **Tiles** section.

Add Linear System to Existing Response Plot

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 **Linear Analysis** tab, in the **Plots** section, select the linear system from the **Select System** menu. Then select the exiting plot from the **Select Plot** menu.

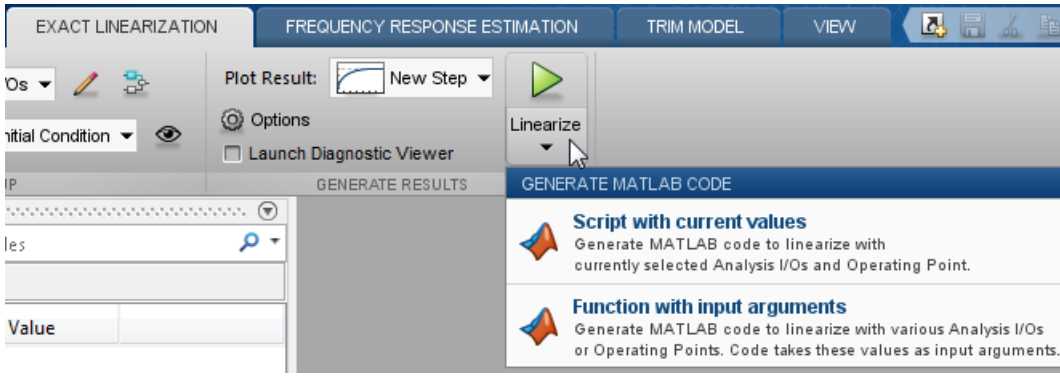
Generate MATLAB Code for Linear Analysis Tool Session

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, on the **Exact Linearization** tab, interactively configure the analysis I/Os, operating point, and other parameters for linearization.

2. Click **Linearize** to open the **Generate Matlab Code** menu.



3. Select the type of code you want to generate:

   - `Script with current values` — 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 with input arguments` — 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`

Related Examples

- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167

Concepts

- "What Is Batch Linearization?" on page 2-160

Troubleshooting Linearization

In this section...

"Linearization Troubleshooting Overview" on page 2-101

"Check Operating Point" on page 2-109

"Check Linearization I/O Points Placement" on page 2-109

"Check Loop Opening Placement" on page 2-110

"Check Phase of Frequency Response for Models with Time Delays" on page 2-110

"Check Individual Block Linearization Values" on page 2-110

"Check Large Models" on page 2-113

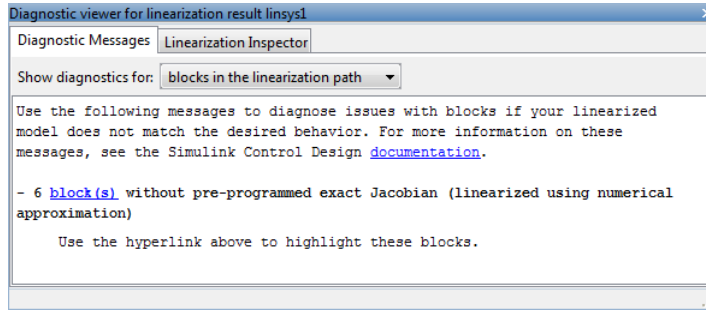
"Check Multirate Models" on page 2-114

Linearization Troubleshooting Overview

- "Troubleshooting Checklist" on page 2-101
- "State-Space, Transfer Function, and Zero-Pole-Gain Equations of Linear Model" on page 2-105
- "Linearization Diagnostics" on page 2-106
- "Highlighting Linearized Blocks" on page 2-108

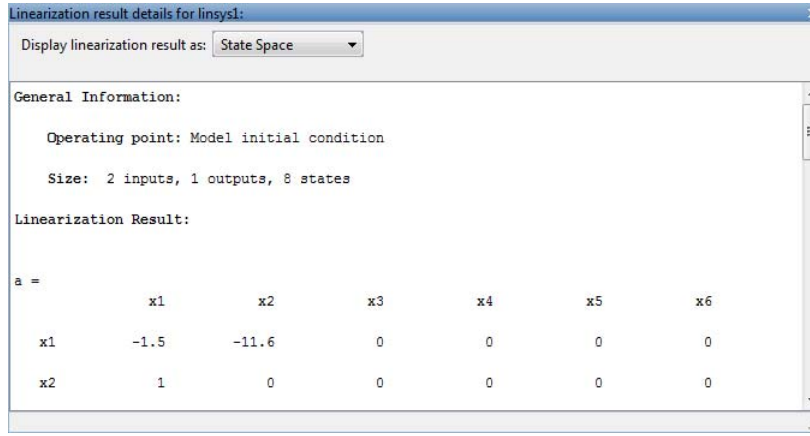
Troubleshooting Checklist

If you do not get good linearization results, use the Linear Analysis Tool's troubleshooting tools. For example, use the **Diagnostic Messages** tab and the **Linearization Inspector** tab that are available in the Diagnostic viewer. To open the Diagnostic viewer, see "Linearization Diagnostics" on page 2-106.



You can also use Linearization result dialog box for troubleshooting. To open the Linearization result dialog box:

1. After linearizing the model, in the **Linear Analysis** tab of the Linear Analysis Tool, select the model of interest in the **Select Result** list.
2. In the **Select Report** list, select **Show result details** to open the Linearization result dialog box for the selected model.



Where to Look	Learn More	Signs of Successful Linearization	Signs of Unsuccessful Linearization
Linear analysis plot, generated after linearization.	Create plots, as described in "Visualize Models" on page 2-96.	Time- or frequency-domain plot characteristics (e.g., rise time, bandwidth) capture system dynamics.	Response plot characteristics do not capture the dynamics of your system. For example, Bode plot gain is too large or too small, or pole-zero plot contains unexpected poles or zeros.
Linear model equations in the Linearization result details dialog box.	View other linear model representations, as described in "State-Space, Transfer Function, and Zero-Pole-Gain Equations of Linear Model" on page 2-105.	State-space matrices show expected number of states. You might see fewer states in the linear model than in your Simulink model because, in many cases, the path between linearization input and output points do not reach all the model states. Poles and zeros are in correct location.	Results show only `D = 0` or `D = Inf`.

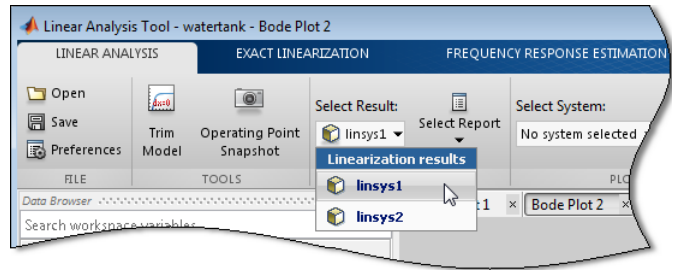
Where to Look	Learn More	Signs of Successful Linearization	Signs of Unsuccessful Linearization
Click in the **Exact Linearization** tab.	"Highlighting Linearized Blocks" on page 2-108	Highlighted blocks in the Simulink model that represent the portion of the model you wanted to linearize.	Missing blocks in the linearization path might indicate incorrect linearization input or output point placement, or that a critical block unexpectedly linearizes to zero, or that critical blocks are connected in a path to a block that linearizes to zero. See "Check Linearization I/O Points Placement" on page 2-109. Extra blocks in the linearization path might indicate incorrect loop opening placement. See "Check Loop Opening Placement" on page 2-110.
Linearization diagnostics in the **Diagnostic Messages** tab of the Diagnostic viewer.	"Linearization Diagnostics" on page 2-106	Message indicates that there are no problematic blocks in the linearization.	One or more warnings about specific problematic blocks. See "Check Individual Block Linearization Values" on page 2-110.

State-Space, Transfer Function, and Zero-Pole-Gain Equations of Linear Model

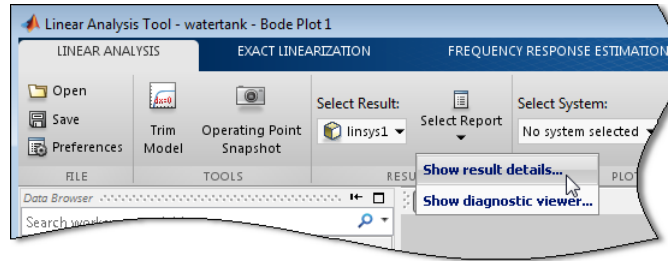
You can view the linear model equations in the Linearization result dialog box of the Linear Analysis Tool.

To open the Linearization result details dialog box:

1. After linearizing the model, in the **Linear Analysis** tab, select the desired linear model from the **Select Result** list.

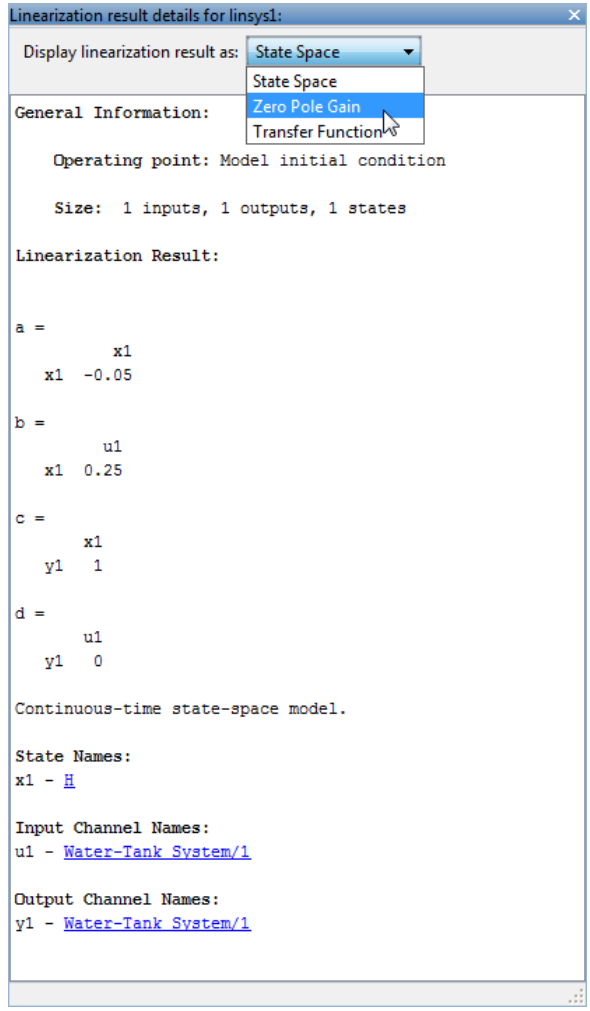


2. Select **Show result details** from the **Select Report** list.



   The Linearization result dialog box opens.

Linear model equations display in state-space form, by default. Alternatively, you can view the Zero Pole Gain or Transfer Function representation in the **Display linearization result as** list.

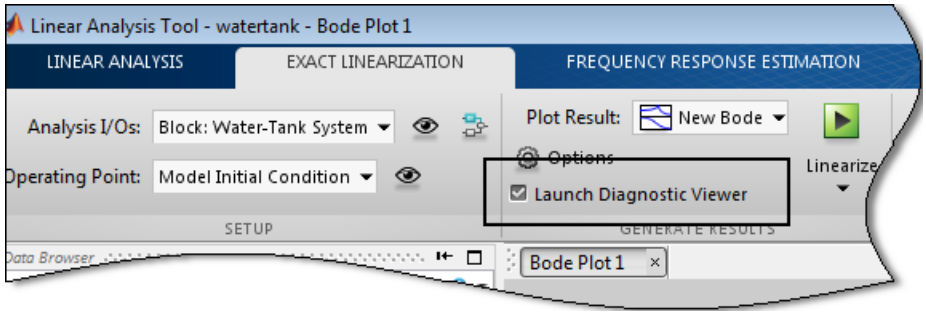


Linearization Diagnostics

You can view diagnostic information about the linearization of specific blocks in the **Diagnostic Messages** tab of the Diagnostic viewer of the Linear Analysis Tool. The **Diagnostic Messages** tab also suggests corrective actions.

Specifically, the **Diagnostic Messages** tab flags blocks with configuration warnings, unsupported blocks, and blocks that automatically linearize using numerical perturbation.

1 In the Linear Analysis Tool, before linearizing the model, select the **Launch Diagnostic Viewer** check box in the **Exact Linearization** tab.



This action ensures that the software logs the diagnostics for the linearized model.

The Diagnostic Messages dialog for the model opens after you linearize the model.

Tip You can also open the Diagnostic viewer for a model by choosing the model in the **Select Result** list in the **Linear Analysis** tab of the Linear Analysis Tool. Then choose **Show diagnostic viewer** in the **Select Report** list.

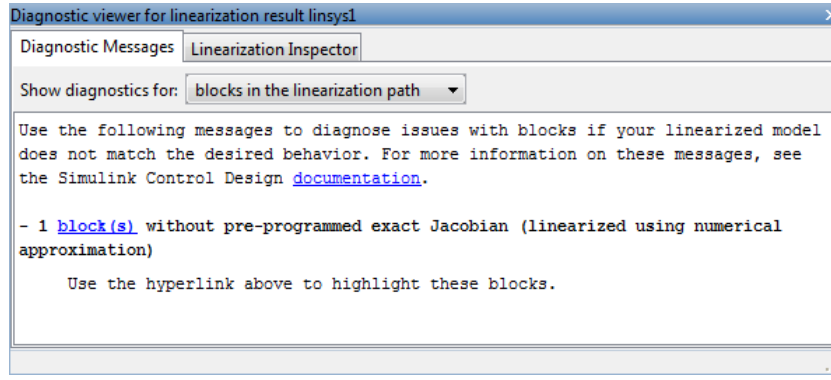
---

2 In the Diagnostic Messages dialog box, the **Show diagnostics for** drop-down list provides the following options:

- `all blocks in the Simulink model` — Use when you suspect that certain blocks are inappropriately excluded from linearization. For example, blocks that linearize to zero (and shouldn't) are excluded from the linearization path.
- `blocks in the linearization path` — Use when you are sure that all of the blocks that should be included in the linearization are included.

That is, your linearization I/O points and any loop openings are set correctly, and blocks do not inappropriately linearize to zero.

In this example, the model contains one block that is linearized using numerical perturbation.



3 To investigate the flagged block, click the block link to highlight the corresponding block in the model.

If the linearization results are poor, you can use the Linearization Inspector to explore linearization values of individual blocks.

Highlighting Linearized Blocks

You can highlight blocks in your model to help you diagnose linearization issues.

To highlight all of the blocks included in the linearization:

- In the Linear Analysis Tool, click in the **Exact Linearization** tab.

  Blocks that linearize to zero do not appear highlighted in the model. Use the Simulink Control Design **Diagnostic Messages** tab to identify the blocks that cause the linearization of certain blocks to be zero.

To remove highlighting in the model:

- In the Simulink model, select **Display> Remove Highlighting**.

More About

- “Check Linearization I/O Points Placement” on page 2-109.
- “Check Loop Opening Placement” on page 2-110.

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 problem, compute a steady-state operating point and repeat the linearization at this operating point.

Related Examples

- “Simulate Simulink Model at Specific Operating Point” on page 1-43
- “Steady-State Operating Points (Trimming) from Specifications” on page 1-14

Check Linearization I/O Points Placement

After linearizing the model, highlight which block are included in the linearization.

Blocks might 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.

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 "Check Individual Block Linearization Values" on page 2-110.

More About

- "Highlighting Linearized Blocks" on page 2-108
- "Specifying Subsystem, Loop, or Block to Linearize" on page 2-13

Check Loop Opening Placement

Incorrect loop opening placement causes unwanted feedback signals in the linearized model.

To fix the problem, use block highlighting 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.

More About

- "Highlighting Linearized Blocks" on page 2-108
- "Opening Feedback Loops" on page 2-14
- "How the Software Treats Loop Openings" on page 2-158

Check Phase of Frequency Response for Models with Time Delays

When the Bode plot shows insufficient lag in phase for a model containing time delays, the cause might be Padé approximation of time delays in your Simulink model.

See "Models with Time Delays" on page 2-127.

Check Individual Block Linearization Values

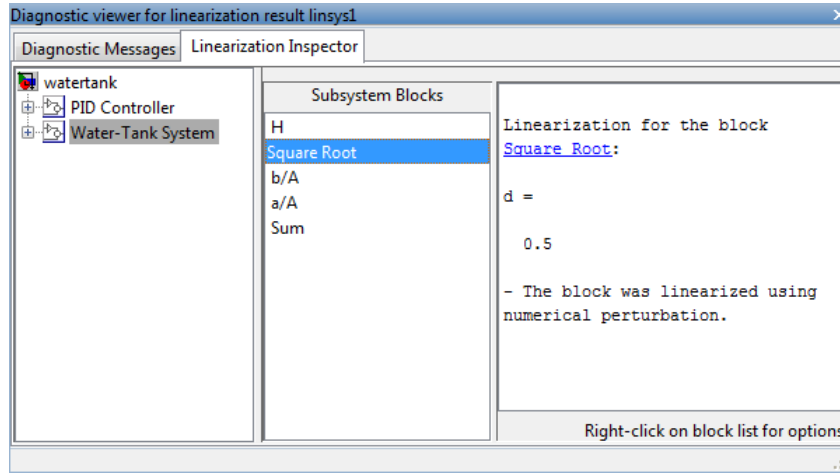
In the Linear Analysis Tool, check the **Diagnostic Messages** tab of the Diagnostic viewer for blocks with configuration warnings, unsupported blocks, and blocks that automatically linearize using numerical perturbation. Click the block link to view the highlighted block in the model.

After identifying the blocks flagged in the **Diagnostic Messages** tab, view the linearization values of these blocks in the **Linearization Inspector** tab of the Diagnostic viewer.

1. In the Linear Analysis Tool, select the linear model for which to display linearization results.
   a. In the **Linear Analysis** tab, use the **Select Result** list to select the model.
   b. In the **Select Report** list, select **Show diagnostic viewer** to open the Diagnostic viewer dialog box.
   c. In the Diagnostic viewer dialog box, click the **Linearization Inspector** tab.
2. Select the specific subsystem or block.

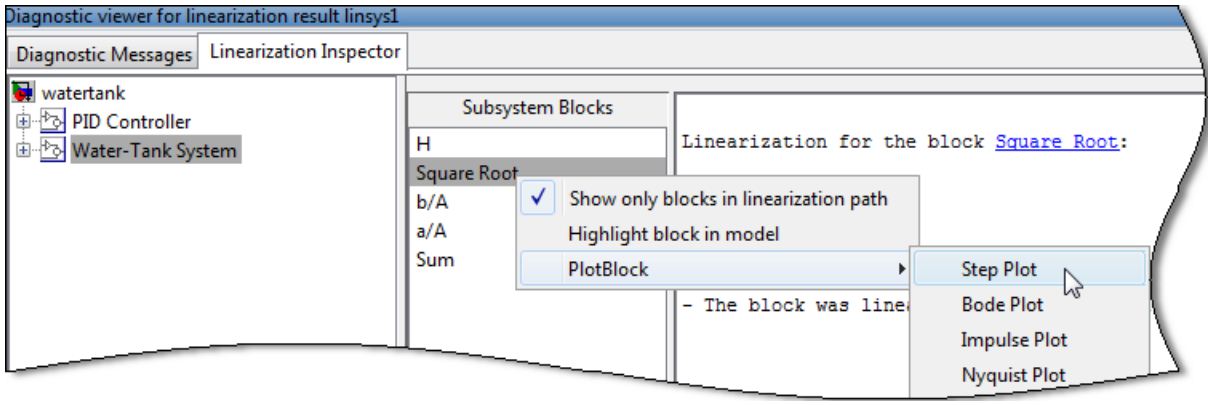
   For example, in the **watertank** model, select the **Water-Tank System** subsystem. Then, in **Subsystem Blocks**, select the **Square Root** block.

Tip Right-click anywhere in the **Subsystem Blocks** list, and select **Show only blocks in linearization path**. This action filters the list to include only the linearized blocks.



3 Plot the response of the linearized block.

For example, right-click the **Square Root** block under **Subsystem Blocks**, and select **PlotBlock > Step Plot**.



The step response of the block is displayed.

4 Troubleshoot individual blocks.

Block or Subsystem Type	Comment	Possible Fix
Incompatible with linearization	Some blocks are implemented without analytic Jacobians and do not support numerical perturbation.	Define custom linearization for affected block as an expression or function. See "Controlling Block Linearization" on page 2-117
Event-based subsystem	Linearization of event-based subsystems is 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. See "Event-Based Subsystems (Externally Scheduled Subsystems)" on page 2-132.

Block or Subsystem Type	Comment	Possible Fix
Simulink blocks in Discontinuities library, such as Deadzone, Saturation, and Quantizer blocks	The Discontinuities library blocks typically have poor linearization results when the operating point that is close to the discontinuity.	If you want the linearization to be a gain of 1, select **Treat as gain when linearizing** in the block parameters dialog box. Define custom linearization for affected block as an expression or function. See "Controlling Block Linearization" on page 2-117.
Model reference block	Linearization is not fully compatible with model reference blocks with Accelerator simulation mode.	Always set each Model (model reference) block to use Normal simulation mode, instead of Accelerator mode.
Blocks that linearize using numerical perturbation, instead of using preprogrammed 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. See "Perturbation Level of Blocks Perturbed During Linearization" on page 2-129.
	Blocks that have nondouble precision inputs signals and states linearize to zero. Use the **Linearization Inspector** tab to view the block linearization.	Convert nondouble-precision data types to double precision. See "Linearizing Blocks with Nondouble Precision Data Type Signals" on page 2-130

Check Large Models

Troubleshooting the linearization of large models is easier using a divide-and-conquer strategy.

Systematically linearize specific model components independently and check whether that component has the expected linearization.

Related Examples

"Plant Linearization" on page 2-33

Check Multirate Models

Incorrect sampling time and rate conversion method can cause poor linearization results in multirate models.

- "Change Sampling Time of Linear Model" on page 2-114
- "Change Linearization Rate Conversion Method" on page 2-115

Change Sampling Time of Linear Model

The sampling time of the linear model displays in the Linearization results dialog box, below the linear equations.

By default, the software chooses the slowest sample time of the multirate model. If the default sampling time is not appropriate, specify a different linearization sample time and repeat.

In Linear Analysis Tool:

1. In the **Exact Linearization** tab, click **Options**.

   The Options for exact linearization dialog box opens.

2. In the **Linearization** tab, enter the desired sample time in the **Enter sample time (sec)** box. Press **Enter**.

   `-1` specifies that the software linearizes at the slowest sample rate in the model.

   `0` specifies a continuous-time linear model.

At the command line, specify the `SampleTime` linearization option.

For example:

```
opt = linearizeOptions;
opt.SampleTime = 0.01;
```

Change Linearization Rate Conversion Method

When you linearize models with multiple sample times, such as a discrete controller with a continuous plant, a rate conversion algorithm generates a single-rate linear model. The rate conversion algorithm affects linearization results.

In the Linear Analysis Tool:

1. Select **Options** in the **Exact Linearization** tab.
2. Click the **Linearization** tab to select the appropriate rate conversion method from the **Choose rate conversion method** list.

Rate Conversion Method	When to Use
Zero-Order Hold	Use when you need exact discretization of continuous dynamics in the time-domain for staircase inputs.
Tustin	Use when you need good frequency-domain matching between a continuous-time system and the corresponding discretized system, or between an original system and the resampled system.
Tustin with Prewarping	Use when you need good frequency domain matching at a particular frequency between the continuous-time system and the corresponding discretized system, or between an original system and the resampled system.
Upsampling when possible (Zero-Order Hold, Tustin, and Tustin with Prewarping)	Upsample discrete states when possible to ensure gain and phase matching of upsampled dynamics. You can only upsample when the new sample time is an integer multiple of the sampling time of the original system. Otherwise,

Rate Conversion Method	When to Use
	the software uses an alternate rate conversion method.

At the command line, specify the `RateConversionMethod` linearization option.

For example:

```
opt = linearizeOptions;
opt.RateConversionMethod = 'tustin';
```

See Also

`linearizeOptions`

Related Examples

Linearization of Multirate Models

Linearization Using Different Rate Conversion Methods

Controlling Block Linearization

In this section...

"When You Need to Specify Linearization for Individual Blocks" on page 2-117

"Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-118

"Specify D-Matrix System for Block Linearization Using Function" on page 2-119

"Augment the Linearization of a Block" on page 2-123

"Models with Time Delays" on page 2-127

"Perturbation Level of Blocks Perturbed During Linearization" on page 2-129

"Linearizing Blocks with Nondouble Precision Data Type Signals" on page 2-130

"Event-Based Subsystems (Externally Scheduled Subsystems)" on page 2-132

When You Need to Specify Linearization for Individual Blocks

Simulink blocks with sharp discontinuities produce poor linearization results. Typically, you must specify a custom linearization for such blocks.

When your model operates in a region away from the point of discontinuity, the linearization is zero. A block with discontinuity linearizing to zero can cause the linearization of the system to be zero when this block multiplies other blocks.

For other types of blocks, you can specify the block linearization as a:

- Linear model in the form of a D-matrix
- Control System Toolbox model object

- Robust Control Toolbox uncertain state space or uncertain real object (requires Robust Control Toolbox software)

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**.

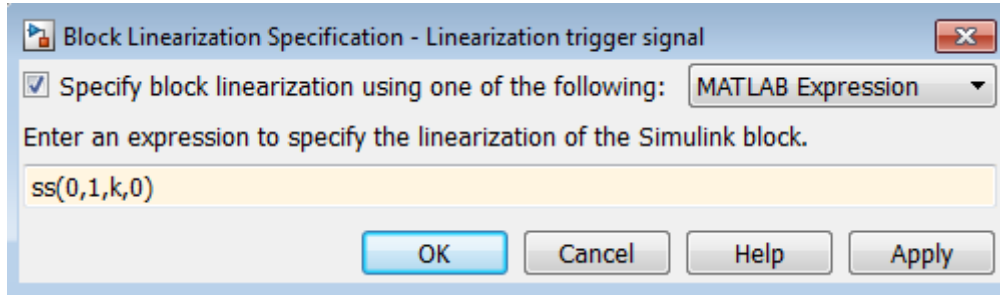
   The Block Linearization Specification dialog box opens.

2. In the **Specify block linearization using a** 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**.

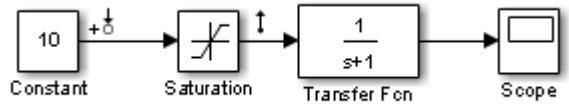
4. Linearize the model.

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.

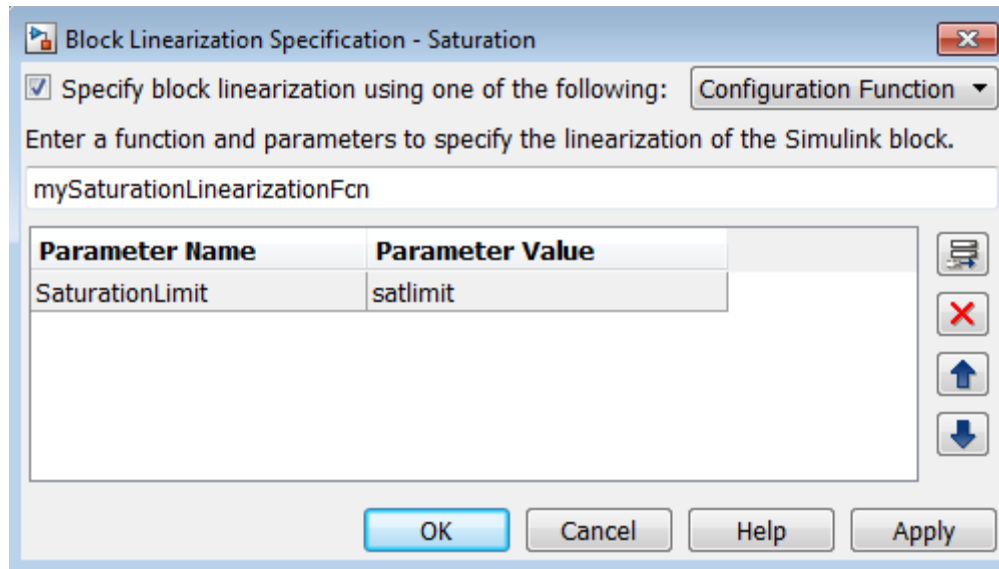
   Configure the linearization function:

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.
```

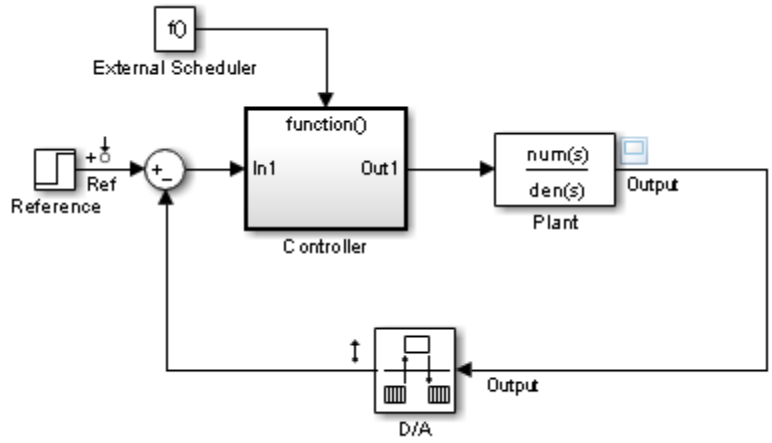
At the model operating point, the input to the saturation block is 10. Therefore, the block linearizes to 0.5, the linearization value specified in the function for saturation boundary.

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.

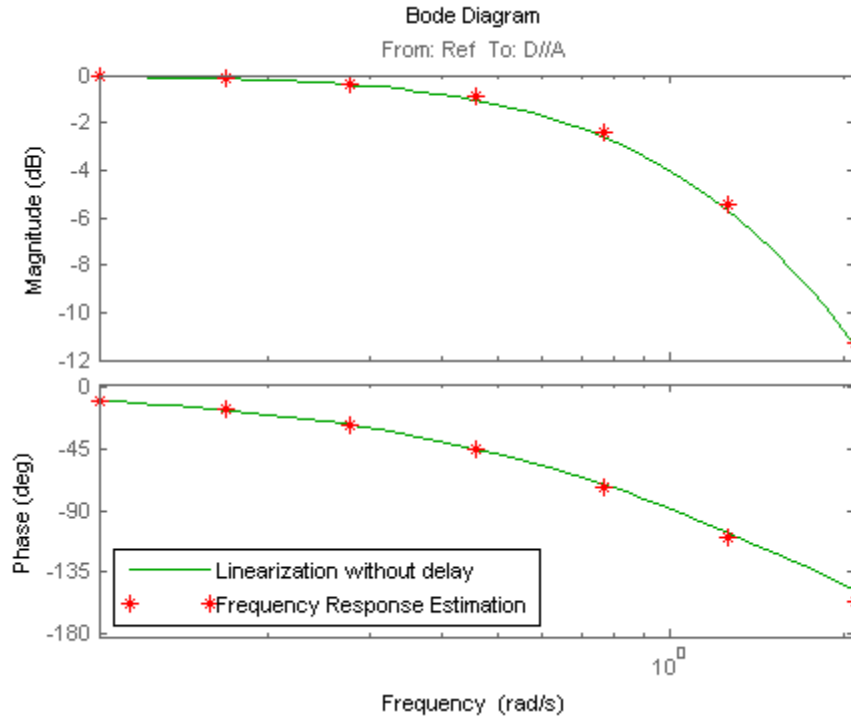
2 (Optional) Linearize the closed-loop model at the model operating point without specifying a linearization for the `Controller` block.

```
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).

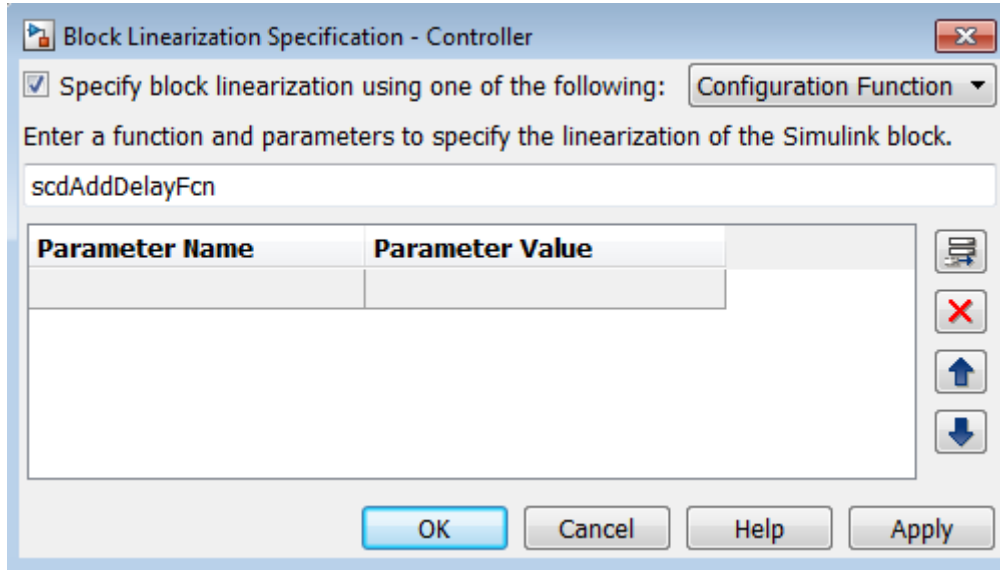
5 Specify the configuration function `scdAddDelayFcn` as the linearization for the `Controller` block.

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**.



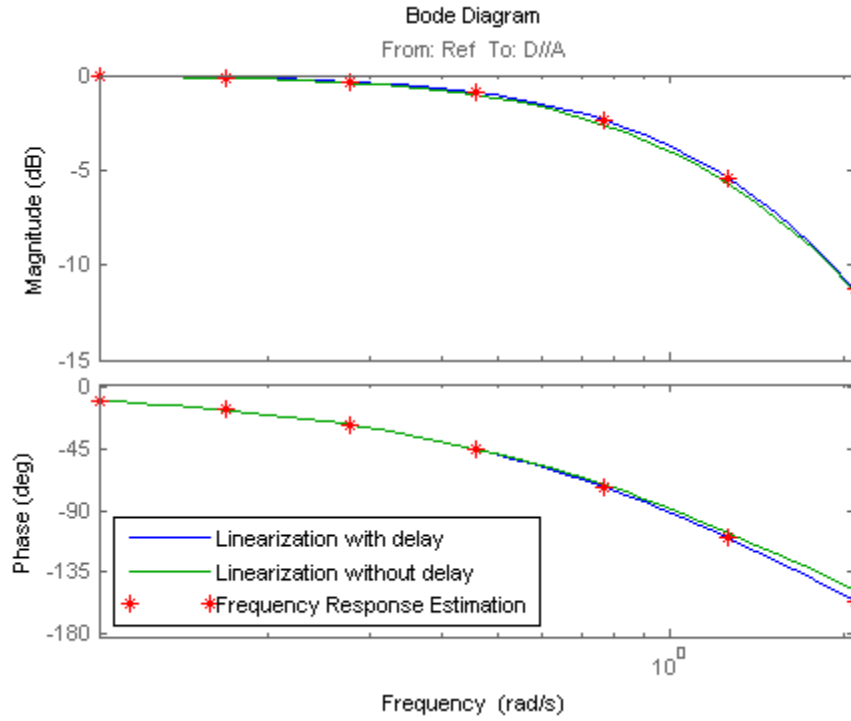
6 Linearize the model using the specified block linearization.

```
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.

Models with Time Delays

- "Choosing Approximate Versus Exact Time Delays" on page 2-127
- "Specifying Exact Representation of Time Delays" on page 2-128

Choosing 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.

Specifying Exact Representation of Time Delays

Before linearizing your model:

- In the Linear Analysis Tool:

  1 In the **Exact Linearization** tab, click **Options** .

    This action opens the Options for exact linearization dialog box.

2. In the **Linearization** tab, select the **Return linear model with exact delay(s)** check box.

- At the command line:

  Use `linearizeOptions` to specify the `UseExactDelayModel` option.

Related Examples

Linearizing Models with Delays

More About

- "Models with Time Delays" in the Control System Toolbox documentation
- "Time-Delay Approximation" in the Control System Toolbox documentation

Perturbation Level of Blocks Perturbed During Linearization

Blocks that do not have preprogrammed analytic Jacobians linearize using numerical perturbation.

- "Change Block Perturbation Level" on page 2-129
- "Perturbation Levels of Integer Valued Blocks" on page 2-130

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 perturbations 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).

Linearizing 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.

- "Overriding Data Types Using Data Type Conversion Block" on page 2-131
- "Overriding Data Types Using Fixed Point Tool" on page 2-132

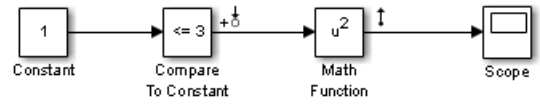
Overriding Data Types Using Data Type Conversion Block

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.

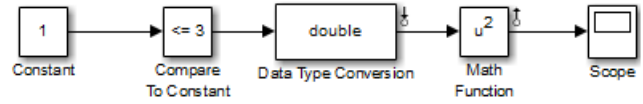
After linearizing the model, remove the Data Type Conversion block from your model.

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.

Event-Based Subsystems (Externally Scheduled Subsystems)

- "Linearizing Event-Based Subsystems" on page 2-133
- "Approaches for Linearizing Event-Based Subsystems" on page 2-133

- "Periodic Function Call Subsystems for Modeling Event-Based Subsystems" on page 2-134
- "Approximating Event-Based Subsystems Using Curve Fitting (Lump-Average Model)" on page 2-137

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 3-28.

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)
```

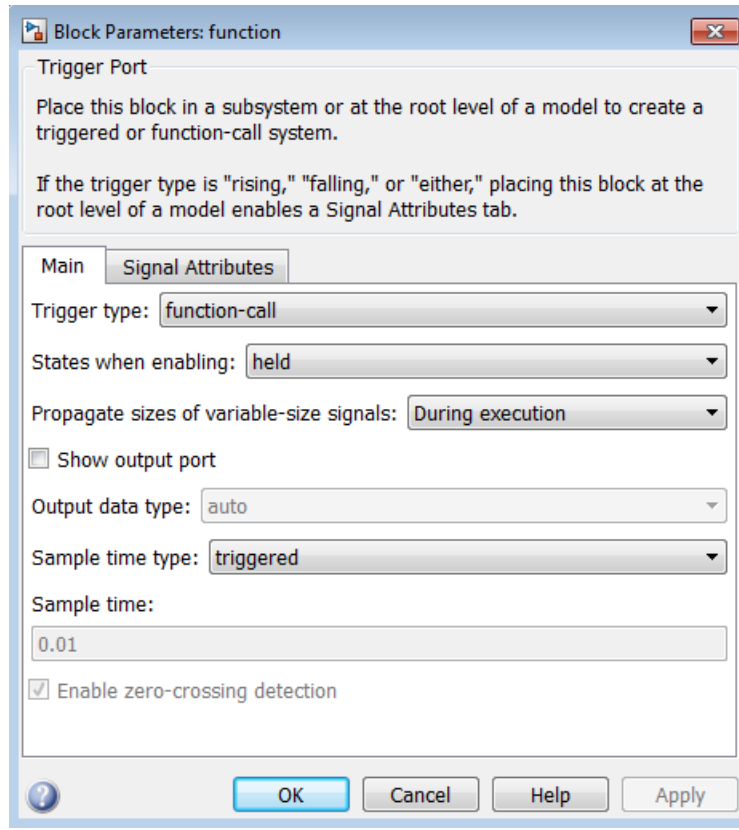
The linearization is zero because the subsystem is not a periodic function call.

```
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.

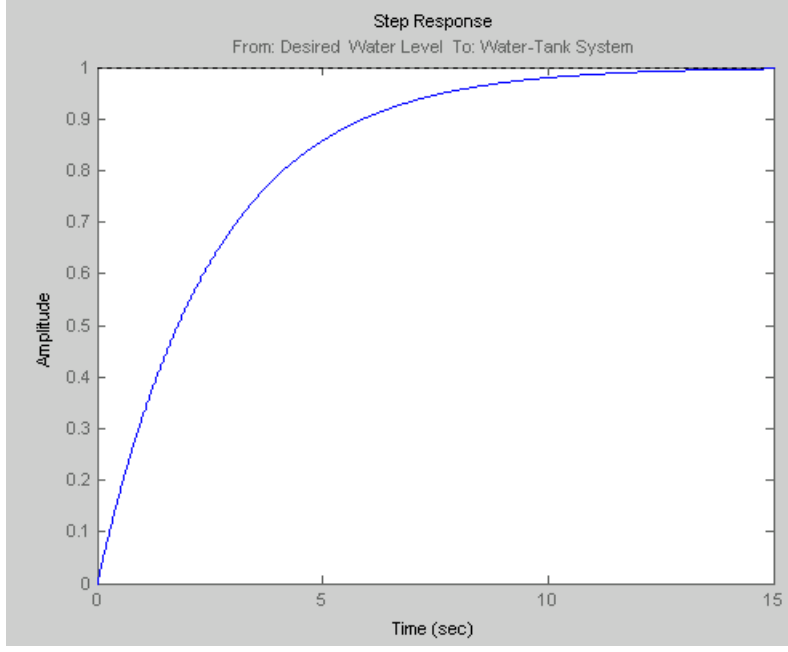
5 Linearize the model.

```
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)
```



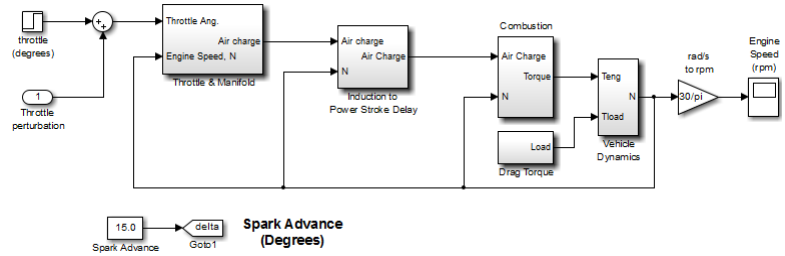
7 Close the model.

```
bdclose(sys);
```

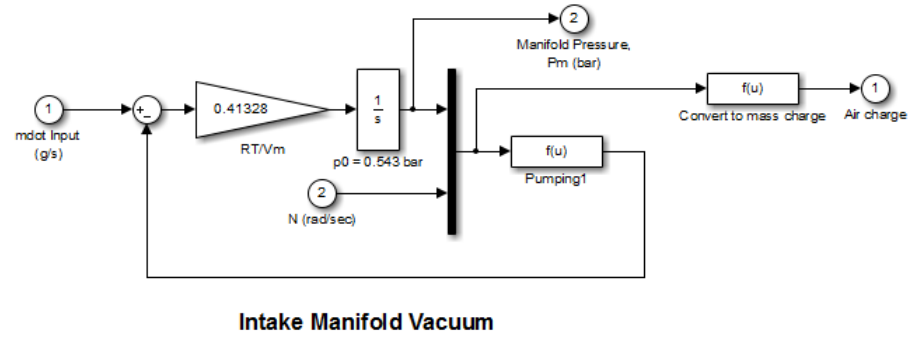
Approximating Event-Based Subsystems Using Curve Fitting (Lump-Average Model)

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.



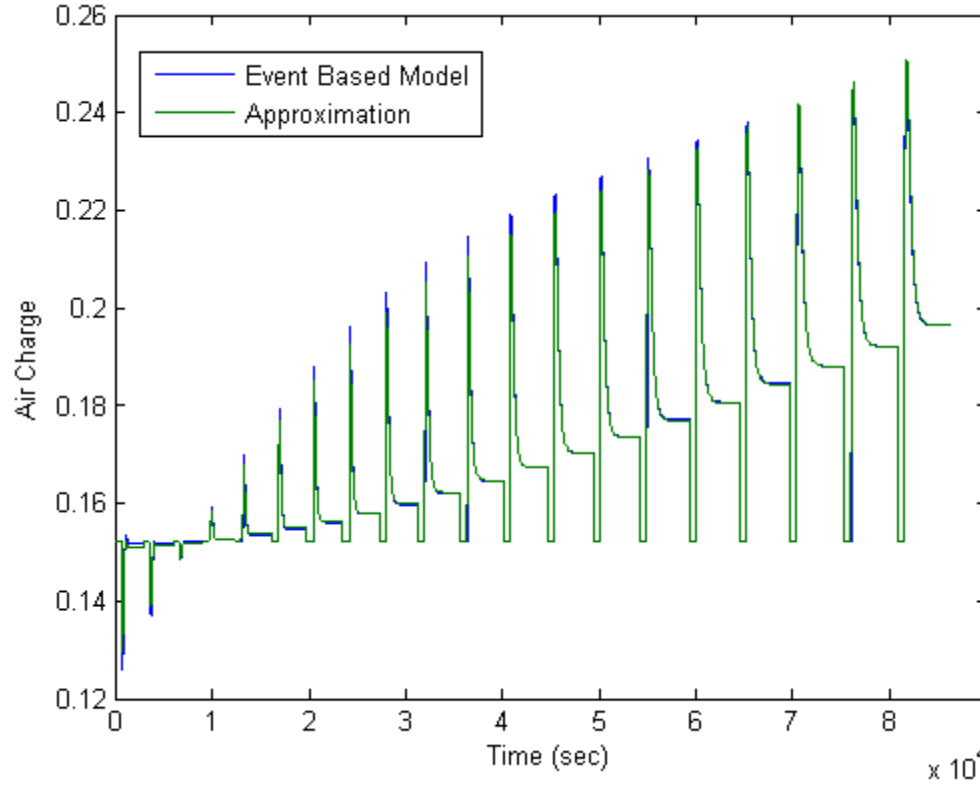
The Convert to mass charge block approximates the relationship between Air Charge, Manifold Pressure, and Engine Speed as a quadratic polynomial.

$$Air\ Charge = p_1 \times Engine\ Speed + p_2 \times Manifold\ Pressure + p_3 \times (Manifold\ Pressure)^2 + p_4 \times Manifold\ Pressure \times Engine\ Speed + p_5$$

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 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.


Event Based Model
Approximation
Air Charge
Time (sec)
x 10^4
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0
1
2
3
4
5
6
7
8
9

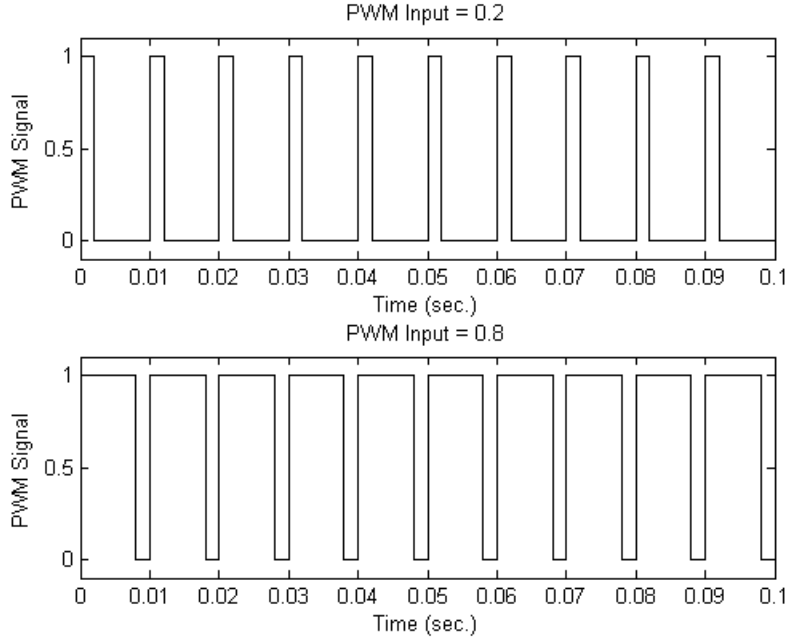
Models with Pulse Width Modulation (PWM) Signals

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.

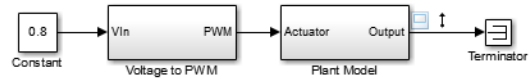
Many industrial applications use Pulse Width Modulation (PWM) signals because such signals are robust in the presence of noise.

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.

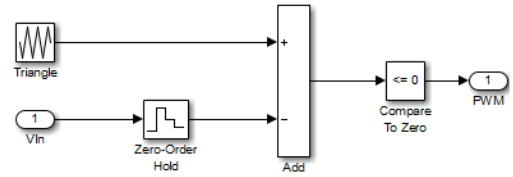


For example, in the `scdpwm` model, a PWM signal is converted to a constant signal.



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.



Related Examples

Specifying Custom Linearizations for Simulink Blocks

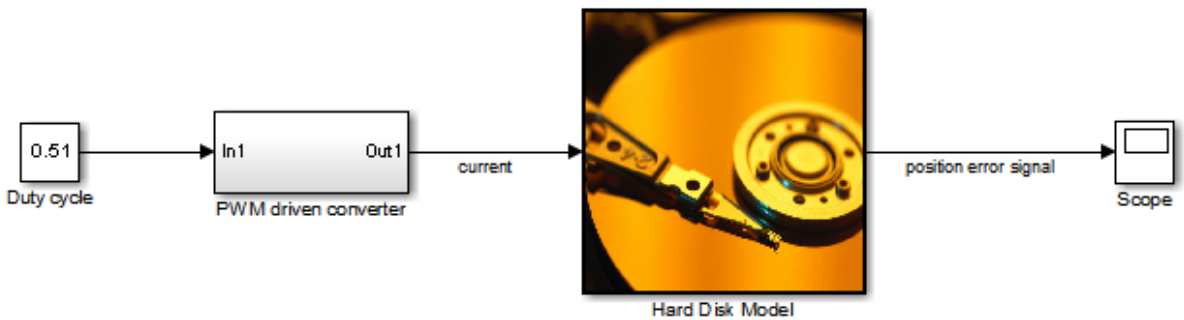
Specifying Linearization for Model Components Using System Identification

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 SimPowerSystems 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.

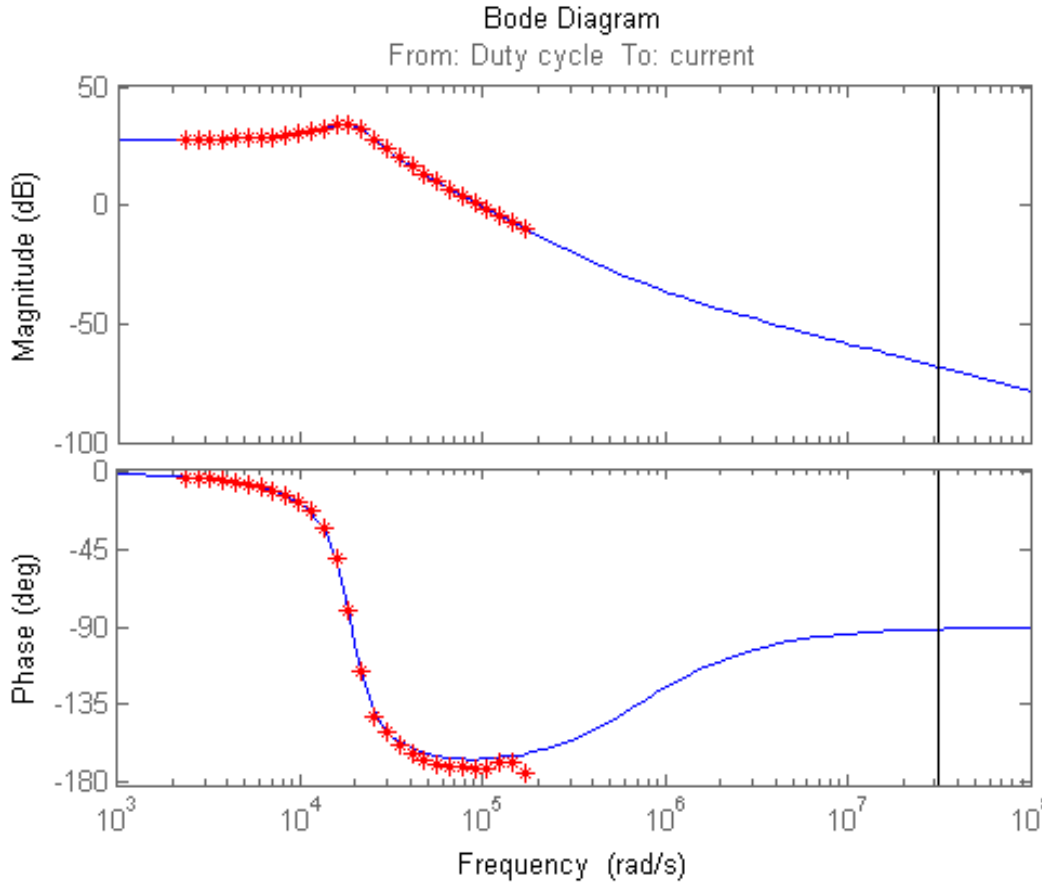
```
idinput = frest.createFixedTsSinestream(Ts,{2000,200000});
idinput.Amplitude = 0.1;
```

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
```

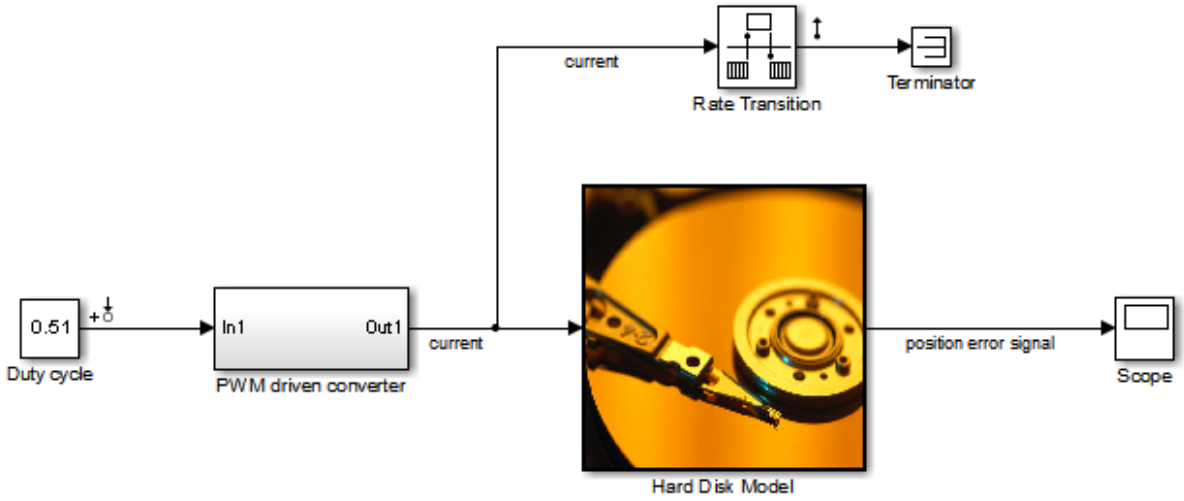
In the input signal parameters, we can see that having a very fast sample rate of 1e-7 seconds (10 MHz sampling frequency) for the frequencies 20 and 200 rad/s cause high `SamplesPerPeriod` values of 3141593 and 314159. Considering that each frequency has 4 periods, frequency response estimation would 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;
```

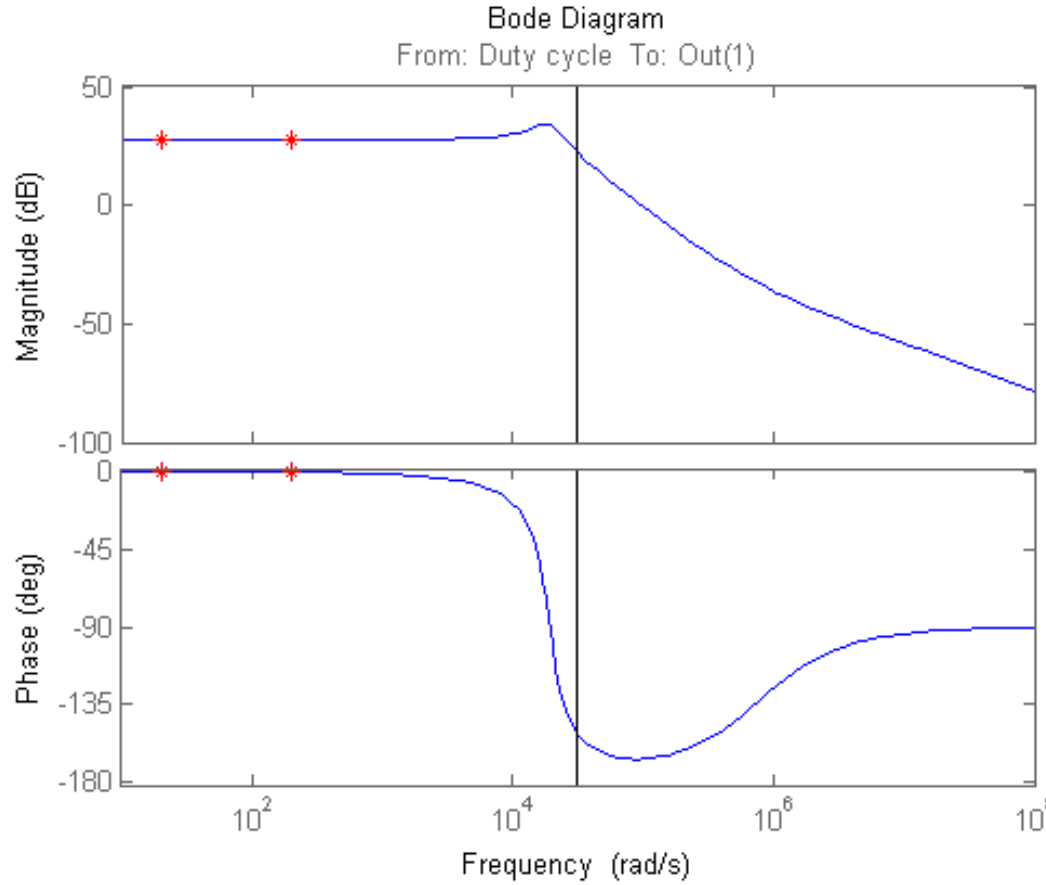
To make the model compatible with the smaller sampling rate, resample the output data point using a rate transition block as in the modified model:

```
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',...
```

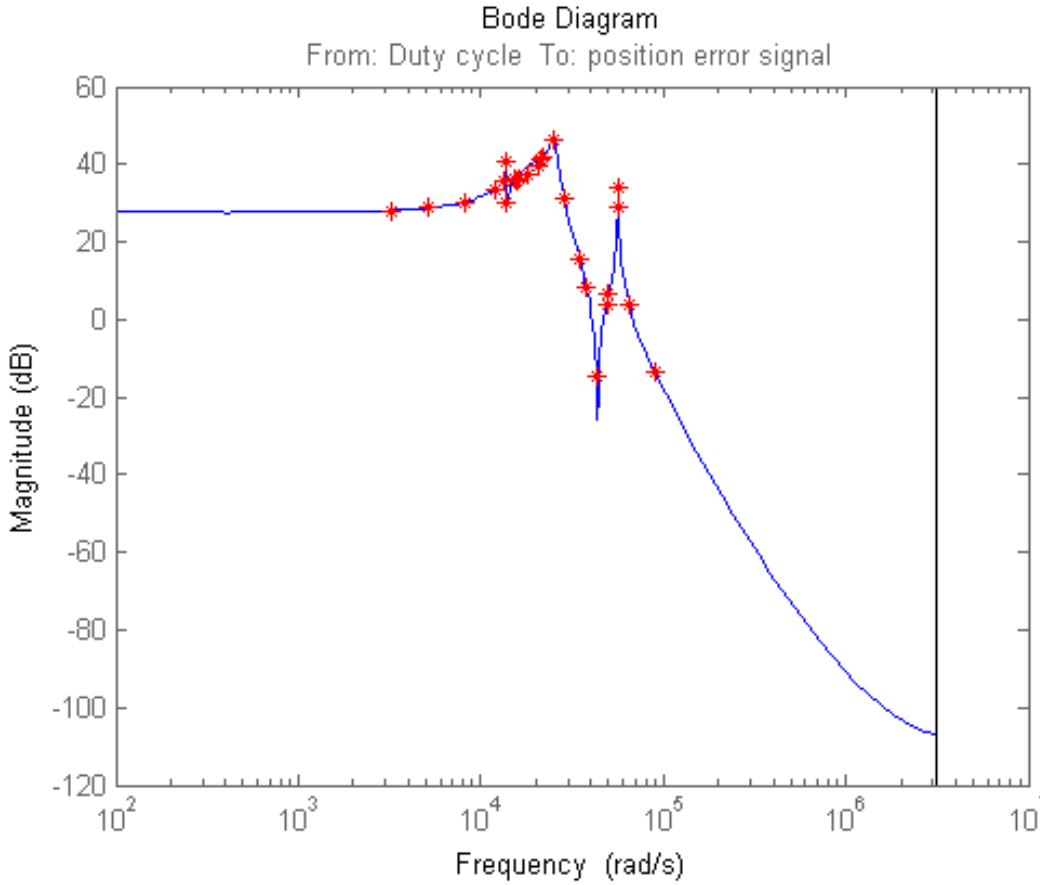
```
                'Type','Expression',...
                'ParameterNames','',...
                'ParameterValues','');
set_param(pwmblock,'SCDBlockLinearizationSpecification',rep);
set_param('scdpwmharddrive/Duty cycle','SampleTime','Ts_plant');
```

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');
```

Speeding Up Linearization of Complex Models

In this section...

"Factors That Impact Linearization Performance" on page 2-150

"Blocks with Complex Initialization Functions" on page 2-150

"Disabling the Linearization Inspector in the Linear Analysis Tool" on page 2-150

"Batch Linearization of Large Simulink Models" on page 2-151

Factors That Impact Linearization Performance

Large Simulink models and blocks with complex initialization functions can slow linearization.

In most cases, the time it takes to linearize a model is directly related to the time it takes to update the block diagram.

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 Inspector in the Linear Analysis Tool

You can speed up the linearization of large models by disabling the Linearization Diagnostics Viewer in the Linear Analysis Tool.

The Linearization Diagnostic Viewer stores and tracks linearization values of individual blocks, which can impact linearization performance.

In the Linear Analysis Tool, in the **Exact Linearization** tab, clear the **Launch Diagnostic Viewer** check box.

Tip Alternatively, you can disable the Linearization Diagnostic Viewer globally in the Simulink Control Design tab of the MATLAB preferences dialog box. Clear the **Launch diagnostic viewer 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 Computing Multiple Linearizations of Models with Block Variations More Efficiently.

Exact Linearization Algorithm

In this section...
"Continuous-Time Models" on page 2-152
"Multirate Models" on page 2-153
"Perturbation of Individual Blocks" on page 2-154
"User-Defined Blocks" on page 2-156
"Look Up Tables" on page 2-157

Continuous-Time Models

Simulink Control Design 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 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:

$$\begin{aligned}
\dot{x}(t) &= f\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
x_1(k_1+1) &= f_1\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
&\vdots \qquad\qquad\qquad \vdots \\
x_m(k_m+1) &= f_i\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right) \\
y(t) &= g\left(x(t), x_1(k_1), \ldots, x_m(k_m), u(t), t\right)
\end{aligned}$$

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 linearizes blocks that do not have preprogrammed linearization using numerical perturbation. The software computes 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.

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.

$$A(:,i) = \frac{\dot{x}|_{x_{p,i}} - \dot{x}_o}{x_{p,i} - x_o}, \qquad B(:,i) = \frac{\dot{x}|_{u_{p,i}} - \dot{x}_o}{u_{p,i} - u_o}$$

$$C(:,i) = \frac{y|_{x_{p,i}} - y_o}{x_{p,i} - x_o}, \qquad D(:,i) = \frac{y|_{u_{p,i}} - y_o}{u_{p,i} - u_o}$$

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.
- $\dot{x}|_{x_{p,i}}$ is the value of $\dot{x}$ at $x_{p,i}$, u_o.
- $\dot{x}|_{u_{p,i}}$ 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.
- $y|_{x_{p,i}}$ is the value of y at $x_{p,i}$, u_o.
- $y|_{u_{p,i}}$ 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.

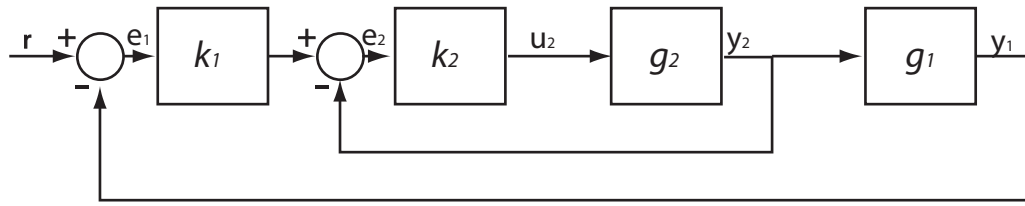
See “Linearizing Blocks with Nondouble Precision Data Type Signals” on page 2-130.

Look Up Tables

Regular look up tables are numerically perturbed. Pre-lookup tables have a preprogrammed (exact) block-by-block linearization.

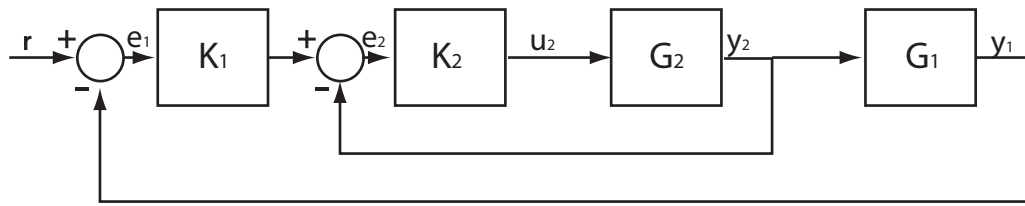
How the Software Treats Loop Openings

To obtain an open-loop transfer function from a model, you specify a loop opening. Loop openings affect only how the software recombines linearized blocks, not how the software linearizes each block. In other words, the software ignores openings when determining the input signal levels into each block, which influences how nonlinear blocks are linearized. Consider the following model, where you obtain the transfer function from `e2` to `y2`, with the outer-loop open at `y1`:



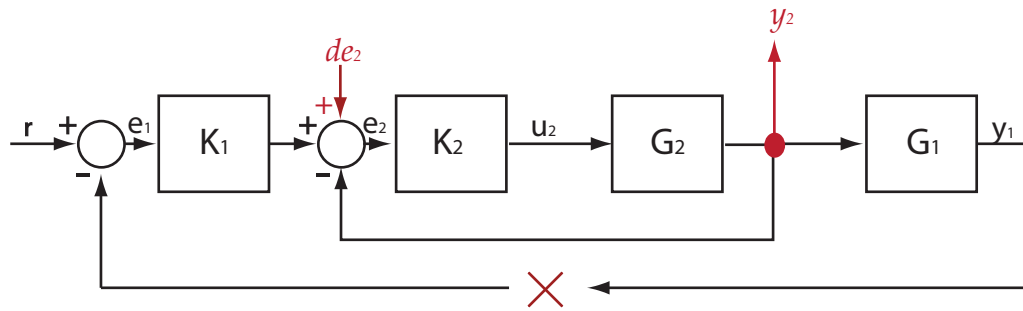
Here, k_1, k_2, g_1, and g_2 are nonlinear.

The software linearizes each block at the specified operating point. At this stage, the software does not break the signal flow at `y1`. Therefore, the block linearizations include the effects of the inner-loop and outer-loop feedback signals.



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

Finally, to compute the transfer function, the software enforces the loop opening at `y1`, injects an input signal at `e2`, and measures the output at `y2`.



The software returns $(I+G_2K_2)^{-1}G_2K_2$ as the transfer function.

See Also

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

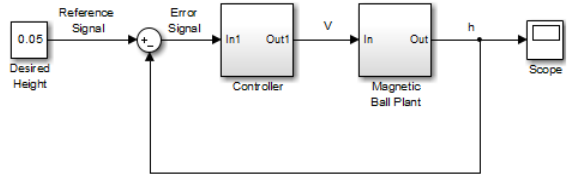
Concepts

- "Opening Feedback Loops" on page 2-14
- "Ways to Specify Portion of Model to Linearize" on page 2-16
- "Exact Linearization Algorithm" on page 2-152

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. You can analyze the time-domain, frequency-domain, and stability characteristics of the linear models obtained by batch linearization. For information regarding the tools that you can use for such analysis, see "Linear Analysis" in the Control System Toolbox documentation.

Consider the magnetic ball levitation model, `magball` (for model details, 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 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`
  - Output disturbance rejection model or sensitivity transfer function, obtained at the outport of `Magnetic Ball Plant` block

- Operating points — Because operating points can influence model dynamics, you linearize a model at different operating points and study their effects on the model. Consider a simple unforced hanging pendulum with angular position and velocity as states. This model has two

steady-state points, one when the pendulum hangs downward, which is *stable*, and another when the pendulum points upward, which is *unstable*. Linearizing this model 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 a variety of ways. For example, you can use 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. 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 impacts the model. 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.

Related Examples

- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169

Concepts

- "Choosing Batch Linearization Tools" on page 2-162

Choosing Batch Linearization Tools

You can perform batch linearization at the command line, using either the `linearize` function or the `slLinearizer` interface. Use the following table to choose a batch linearization tool.

Reasons to Use linearize	Reasons to Use slLinearizer Interface
• You are a new user or have experience in using the Linear Analysis Tool. 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 programmatically reproduce your session, this code uses `linearize`. You can easily modify this code to batch linearize a model. • You are extracting linearizations for only one I/O set (single transfer function).	• 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 use `linearize` to obtain multiple open-loop and closed-loop transfer functions. To do so, you call `linearize` multiple times, specifying a different I/O set for each call. However, the software recompiles the model each time you call `linearize`.

Related Examples

• "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
• "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167
• "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
• "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177

Concepts

- "What Is Batch Linearization?" on page 2-160
- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

Batch Linearization Efficiency When You Vary Parameter Values

You can use the `slLinearizer` interface and the `linearize` command to batch linearize a model by varying model parameter values. When you vary the value of tunable parameters, both tools are efficient, using a single model compilation to compute linearizations for all parameter grid points. *Tunable parameters* refers to parameters whose values you can change during model simulation without recompiling the model. Common tunable parameters include the Gain parameter of the Gain block and Numerator and Denominator coefficients of the Transfer Fcn block.

However, if you vary the value of nontunable parameters, both tools compile the model for each parameter grid point, which makes batch linearization slower. To take advantage of the single model compilation, convert your nontunable parameters to tunable parameters. For example, suppose your model uses the **Inline parameters** option (see "Inline parameters") to optimize the generated code's memory and processing requirements. Before batch linearizing the model, clear this check box to make your model parameters tunable. Some parameters, such as block sample times, cannot be made tunable.

See Also

`slLinearizer | linearize`

Related Examples

- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Specify Parameter Samples" on page 2-190

Concepts

- "About Block Parameters"
- "Model Parameters"

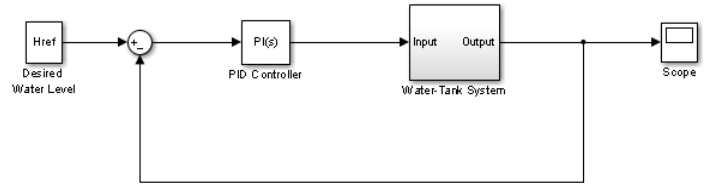
Batch Linearize Model for Parameter Value Variations Using linearize

This example shows how to use the `linearize` command to extract multiple linearizations from a model for varying parameter values (batch linearization).

In this example, you vary the plant parameters and obtain the closed-loop transfer function from the reference input to the plant output for the `watertank` model. You can analyze the batch linearization results, for example, to determine the controller robustness to variations in the plant model.

Open the model.

```
open_system('watertank');
```



Specify the reference input and plant output as the linearization I/Os.

```
io(1) = linio('watertank/Desired  Water Level',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

`io(1)`, the signal originating at the outport of the `Desired Water Level` block, is the reference input. `io(2)`, the signal originating at the outport of the `Water-Tank System` block, is the plant output.

Vary the plant parameters `A` and `b`, used in the `Water-Tank System` block, in the 10% range. `linearize` requires a structure with fields `Name` and `Value` to specify the parameters being varied.

```
[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;
```

`params` specifies a 3 x 4 parameter grid, where each grid point corresponds to a unique combination of `A` and `b` values.

Obtain the closed-loop transfer function from the reference input to the plant output for the specified range of plant parameter values.

```
T = linearize('watertank',io,params);
```

`T` is a 3 x 4 array of linearized models. Each entry in the array contains a linearization for the corresponding parameter combination in the grid specified by `params`. For example, `T(:,:,2,2)` corresponds to the linearization obtained by setting the values of the `A` and `b` parameters to `A_grid(2,2)` and `b_grid(2,2)`.

See Also

`linearize | linio | ndgrid`

Related Examples

- "Specify Parameter Samples" on page 2-190
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183
- "Batch Linearize Model at Multiple Operating Points Using linearize" on page 2-167

Concepts

- "watertank Simulink Model"
- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

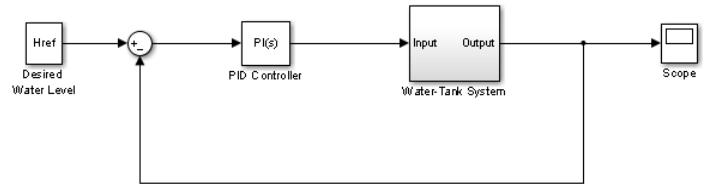
Batch Linearize Model at Multiple Operating Points Using linearize

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)`.

See Also

`linearize | findop | linio`

Related Examples

- "Batch Compute Steady-State Operating Points" on page 1-38
- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "watertank Simulink Model"

Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer

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.

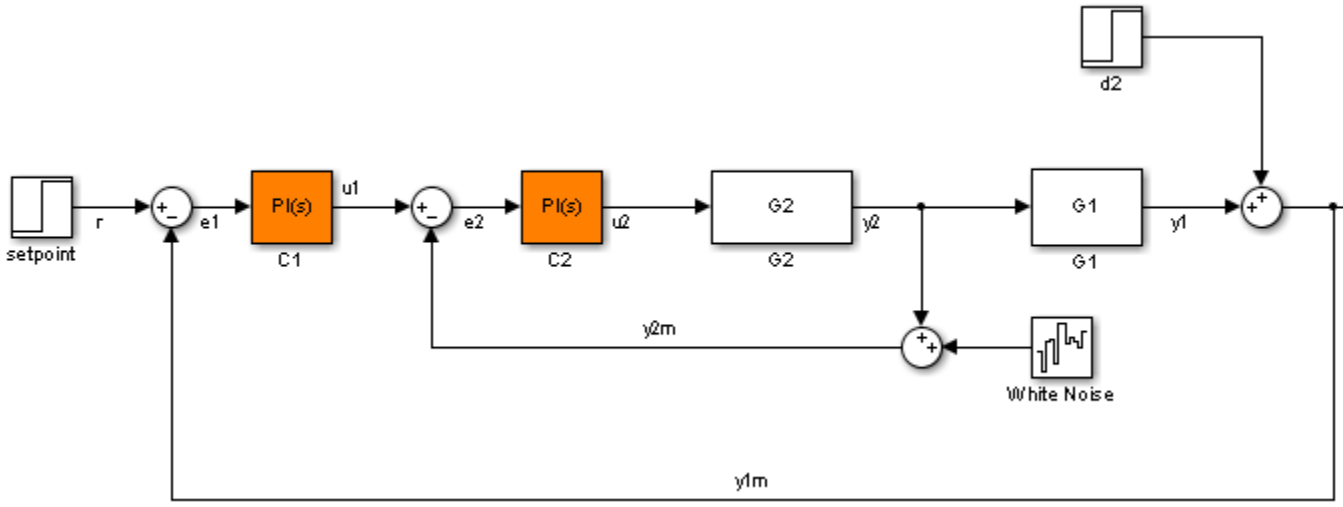
You can perform the same analysis using the `linearize` command. However, when you want to obtain multiple open-loop and closed-loop transfer functions, especially for models that are expensive to compile repeatedly, `slLinearizer` can be more efficient.

Cascaded loop model

The `scdcascade` model 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- and outer-loop dynamics.

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



Create an slLinearizer interface for scdcascade.

```
sllin = slLinearizer(mdl)
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use addPoint method to add new points.
No permanent openings. Use addOpening method to add new permanent openings.
Other properties (with dot notation get/set access):
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Vary the 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 the overall 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`.

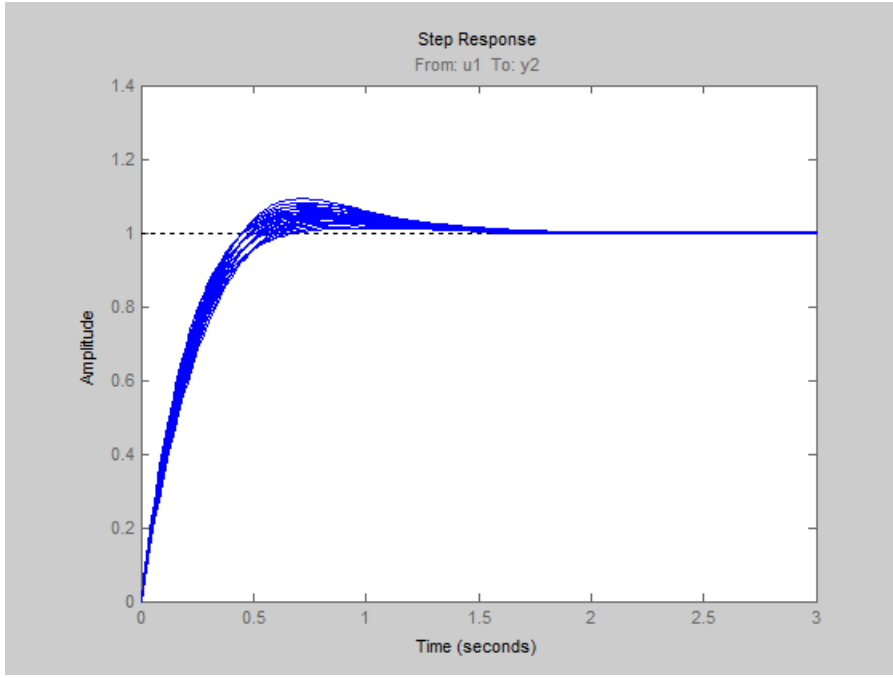
```
r2yi = getIOTransfer(sllin,'u1','y2');
```

`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 the inner-loop transfer function at the plant output.

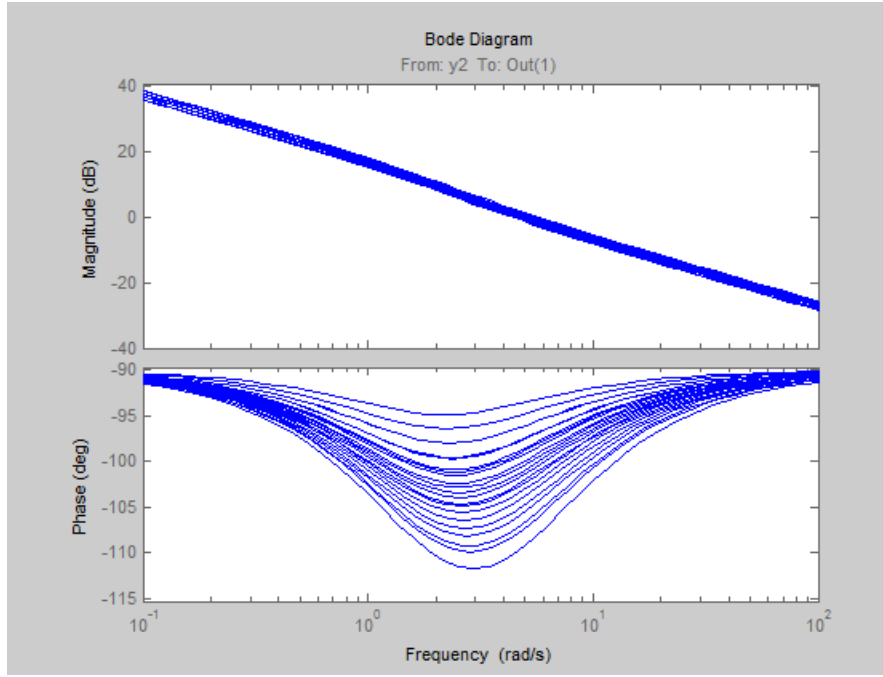
Obtain the inner-loop transfer function at `y2`, with the outer loop open at `e1`.

```
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, $L_i = -G_2C_2$.

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 the 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 the closed-loop transfer function from the reference to the 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`.

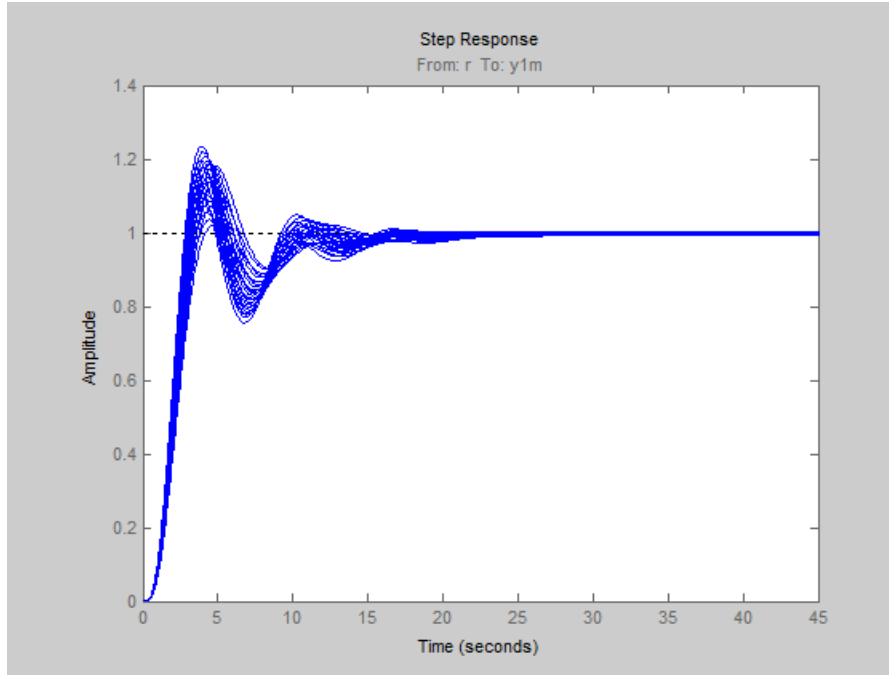
```
addPoint(sllin,{'r','y1m'});
```

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 the outer-loop sensitivity at the plant output.

To obtain the outer-loop sensitivity at the plant output, add `y1` as an analysis point to `sllin`.

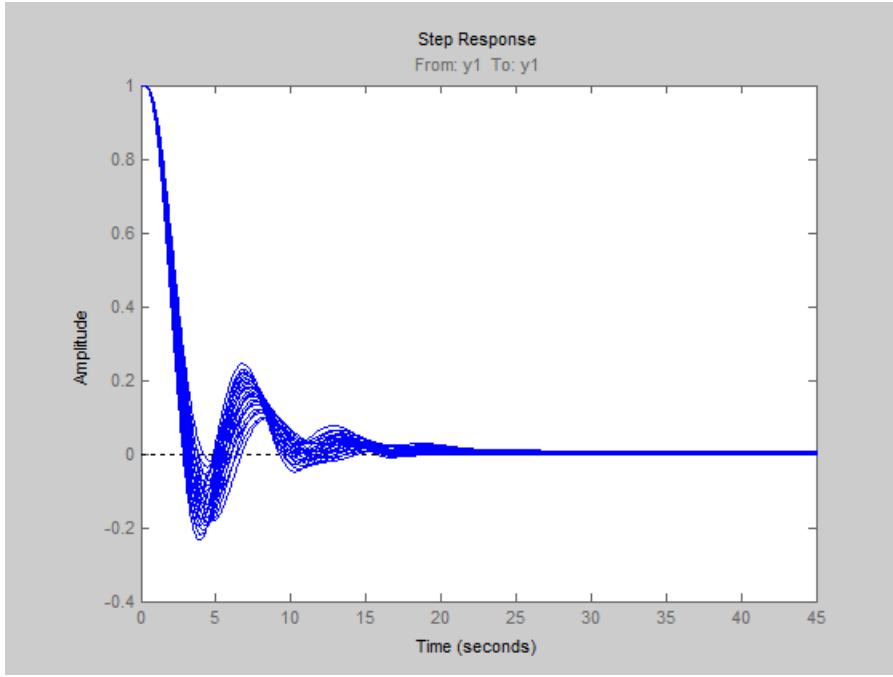
```
addPoint(sllin,'y1');
```

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`.

Close the model.

```
bdclose(mdl);
```

See Also

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

Related Examples

- "Specify Parameter Samples" on page 2-190
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177

Concepts

- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

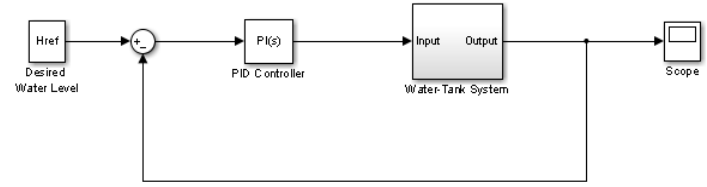
Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer

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.

Open the model.

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



Create an slLinearizer interface for the watertank model.

```
sllin = slLinearizer(mdl)

slLinearizer linearization interface for "watertank":

No analysis points. Use addPoint to add new points.
No permanent openings. Use addOpening 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]
```

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;

h = [10 15 20];

for ct = 1:numel(h)
    opspec.States(2).x = h(ct);
    Href = h(ct);
    ops(ct) = findop(mdl,opspec);
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 the 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 addOpening to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : [1x1 struct], 1 parameters with sampling grid of
            "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.

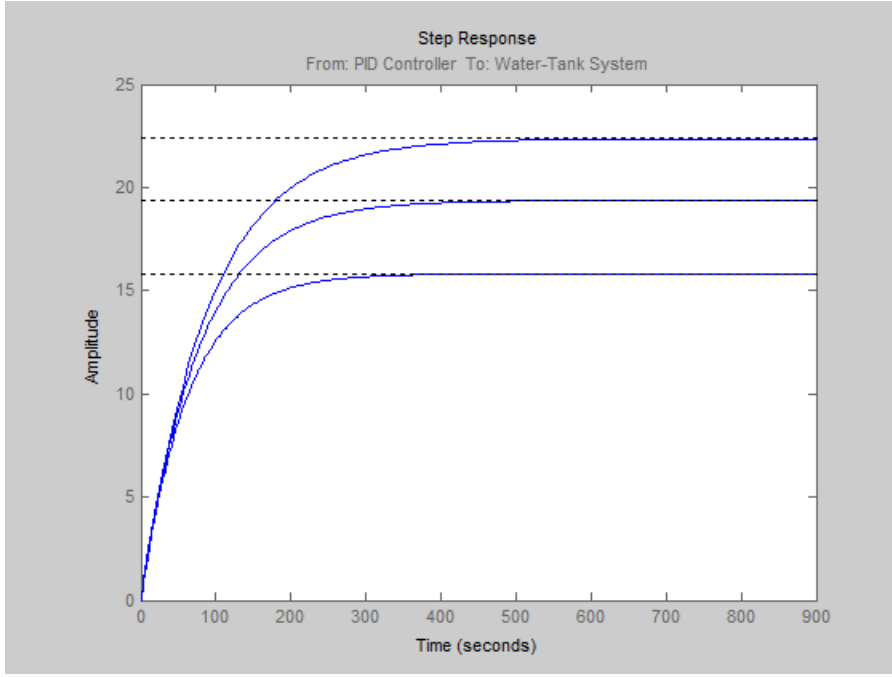
```
G = getIOTransfer(sllin,'PID','Tank','Tank');
```

In the call to `getIOTransfer`, `'PID'`, a substring of the block name `'watertank/PID Controller'`, specifies the first analysis point as the transfer function input. Similarly, `'Tank'`, a substring 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.

Analyze the 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');
```

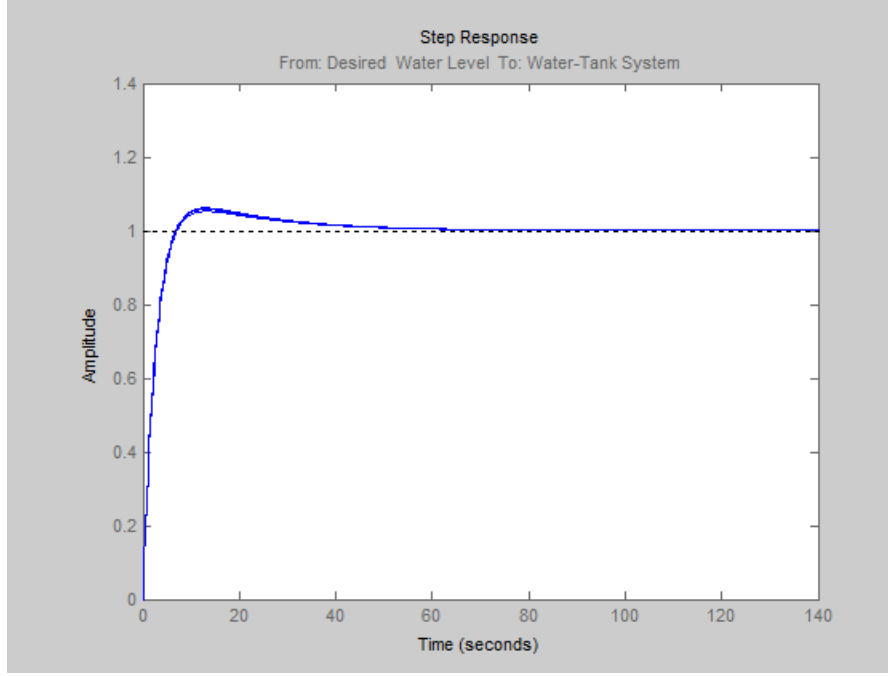
Obtain the closed-loop transfer function.

```
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 two simulation snapshot times, the closed-loop transfer is close.

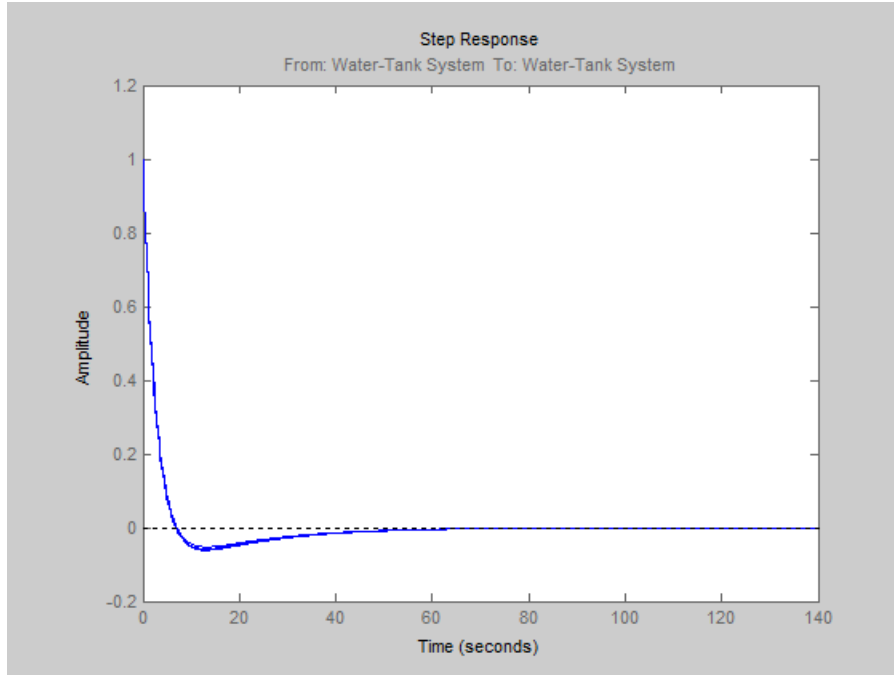
Analyze the sensitivity at the plant output.

```
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

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

Related Examples

- "Batch Compute Steady-State Operating Points" on page 1-38
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "watertank Simulink Model"

Use Response Plots to Analyze Batch Linearization Results

This example shows how to plot and analyze the step response for batch linearization results. *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. You can use the technique in this example to analyze the frequency response, stability, or sensitivity for batch linearization results.

Obtain batch linearization results.

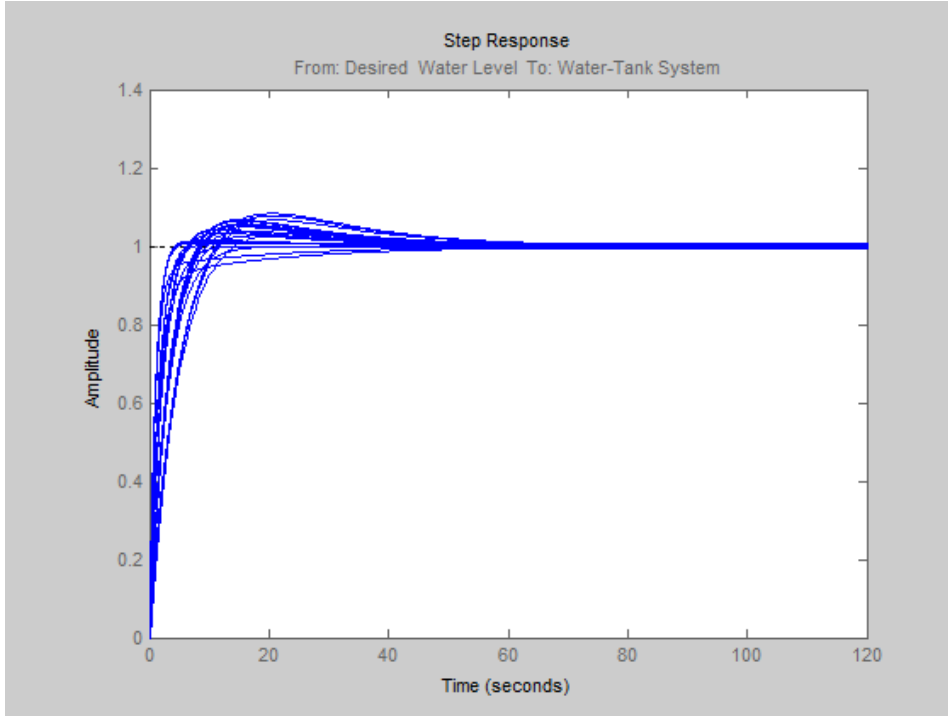
Load the batch linearization results saved in `scd_batch_lin_results1.mat`.

```
load scd_batch_lin_results1 linsys;
```

`linsys`, a 4 x 3 x 2 `ss` model array, contains the closed-loop transfer function of the `watertank` model from the reference input to the plant output. `linsys` was obtained for four simulation snapshot times, `t = [0 1 2 3]`, by varying the model parameters, `A` and `b`. The sample values for `A` are `[10 20 30]`, and the sample values for `b` are `[4 6]`.

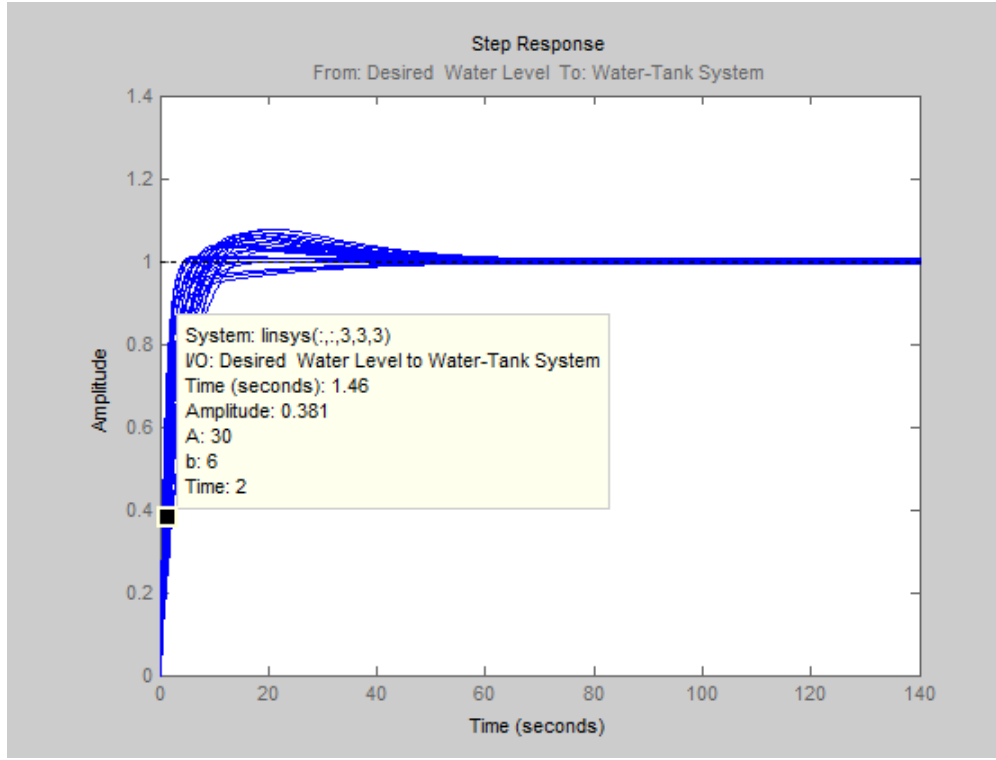
Plot the step response for the linearized models.

```
stepplot(linsys);
```



View the parameter combination and simulation snapshot time that yielded a specific response.

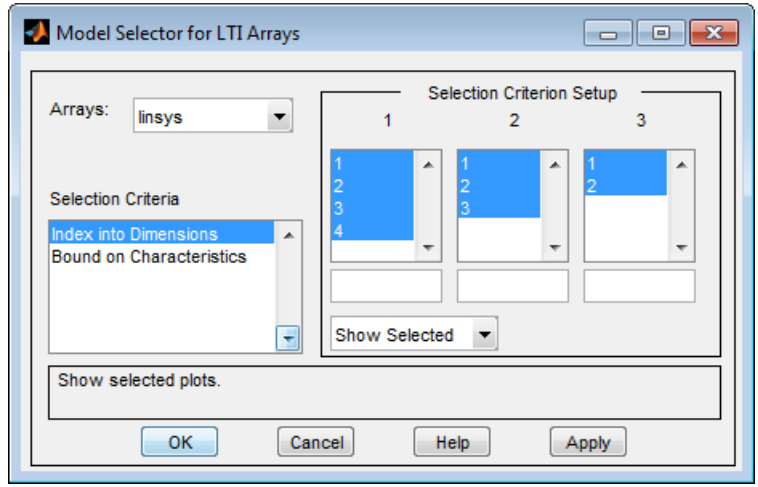
Click the response.



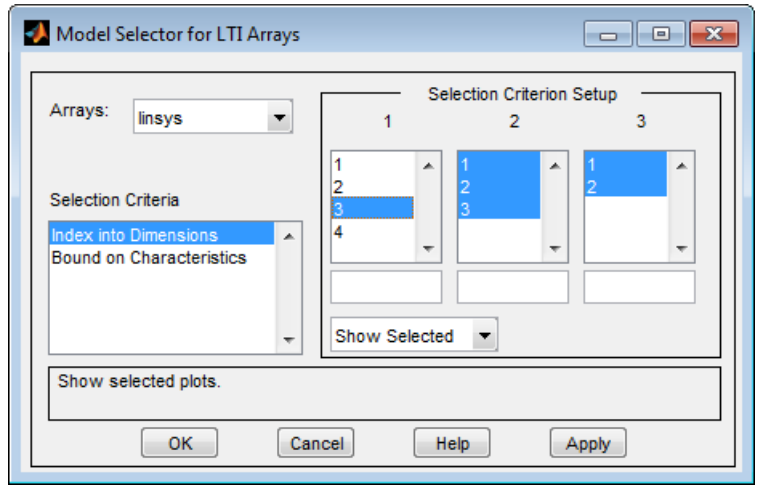
A data tip appears on the plot, providing information about the selected response and the related model. The last lines of the data tip show the parameter combination and simulation snapshot time that yielded this response. In the previous plot, the selected response corresponds to the model obtained by setting `A` to `30` and `b` to `6`. The software linearized the model after simulating the model for two time units.

Plot the step response for a subset of the batch linearization results.

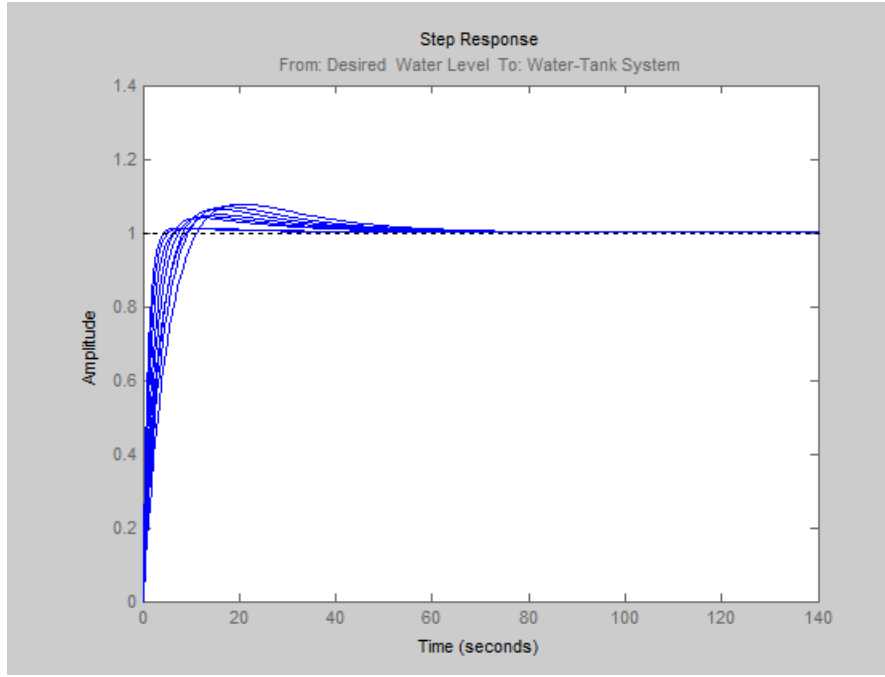
Suppose you want to view the responses for models linearized at a specific simulation snapshot time, for example two time units. Right-click the plot and select **Array Selector**. The Model Selector for LTI Arrays dialog box opens.



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. 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 the step response for a specific parameter combination and simulation snapshot time.

Suppose you want to view 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 the linearized models, which is specified as a structure. When you perform batch linearization, the software populates the `SamplingGrid` structure with information regarding the variable values used to obtain the model. *Variable values* includes 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`.

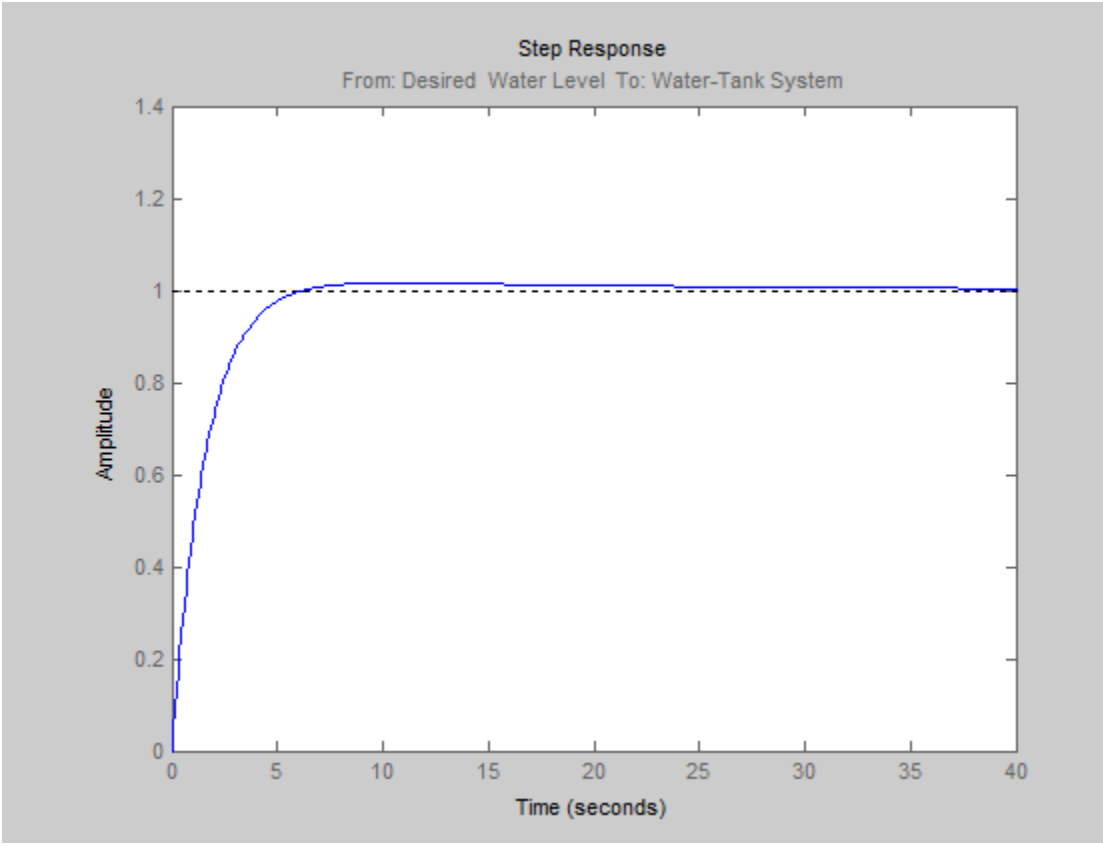
Extract, from `linsys`, the model obtained by linearizing the `watertank` model at `t = 3`, for `A = 10` and `b = 4`:

```
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);
```



Related Examples

- “Validate Batch Linearization Results” on page 2-196

Specify Parameter Samples

In this section...
"Single Parameter" on page 2-190
"Multiple Parameters" on page 2-191

Parameters configure a Simulink model in a variety of ways. For example, you can use 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.

You can use the `slLinearizer` interface or `linearize` command to specify a range of values for a parameter, also referred to as a *parameter grid* or *parameter samples*. The tools batch linearize the model, computing a linearization for each parameter value in the parameter grid. You can also vary multiple parameters, thus extending the parameter grid dimension. Both tools require that you specify the parameter samples using a structure with fields `Name` and `Value`.

Single Parameter

To vary the value of a single parameter, for example a, specify the parameter grid as follows:

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

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

The table lists some common ways of specifying parameter samples.

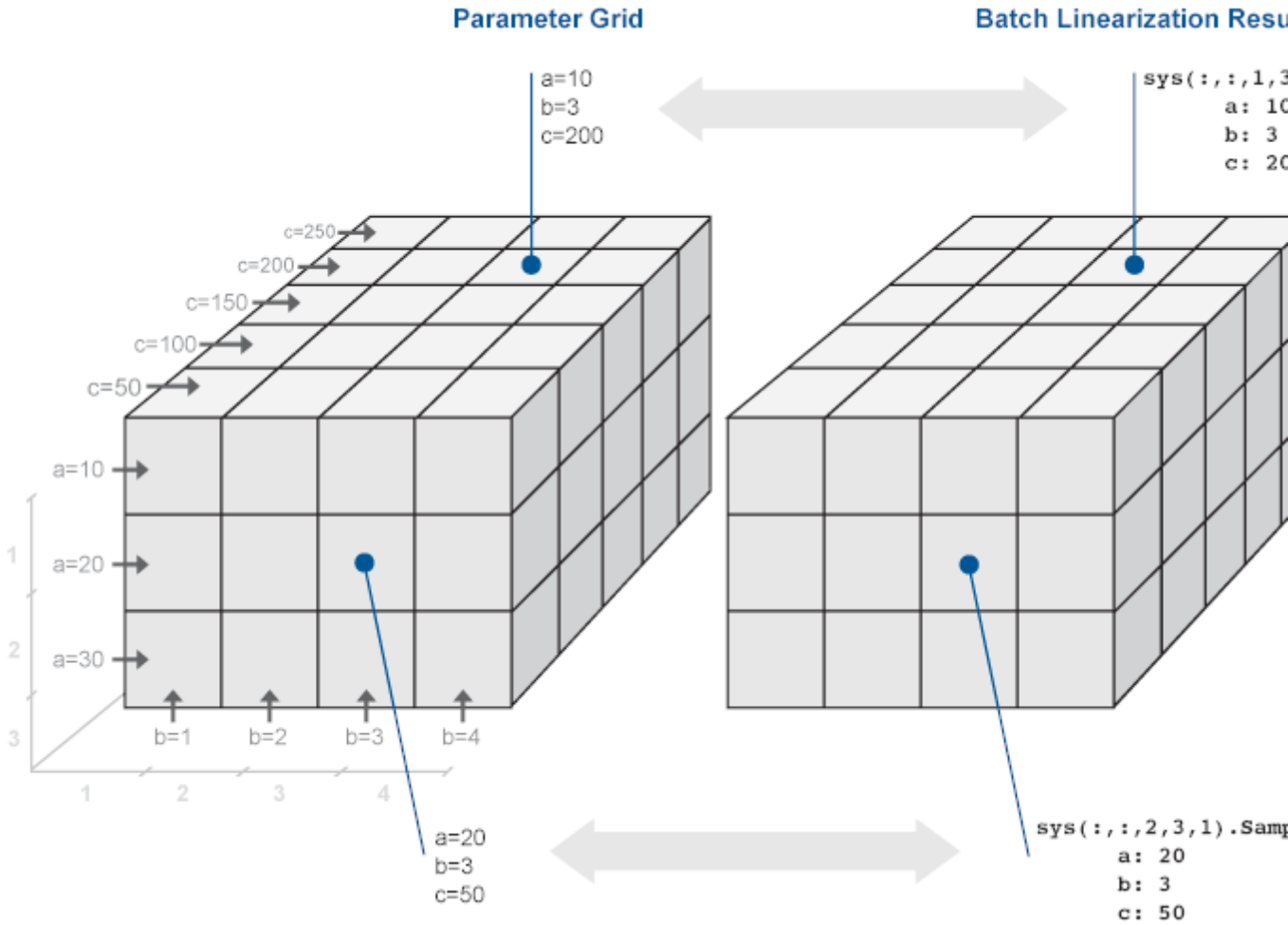
Parameter Sample-Space Type	How to Specify the Parameter Samples
Linearly varying	`param.Value = linspace(p_min,p_max,num_samples);`
Logarithmically varying	`param.Value = logspace(p_min,p_max,num_samples);`

Parameter Sample-Space Type	How to Specify the Parameter Samples
Random	`param.Value = rand(1,num_samples);`
Custom	`param.Value = custom_vector;`

Multiple Parameters

To vary the value of multiple parameters, specify a structure array with an entry for each parameter whose value you vary. The structure for each parameter is the same as described in "Single Parameter" on page 2-190. 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.

Consider this parameter grid:



Here, you vary the values of three parameters, *a*, *b*, and *c*. The samples form a 3 x 4 x 5 grid. The `ss` model array, `sys`, is the batch linearization result.

Specify Full Grid

Suppose 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*:

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

Use `ndgrid` to create a rectangular parameter grid.

```
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 2-192, 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;
```

See Also

`ndgrid` | `linspace` | `logspace` | `rand` | `slLinearizer` | `linearize`

Related Examples

- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169

Concepts

- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

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 (described below), but both methods have some computational overhead. This overhead can make validating each model in the batch linearization results infeasible. Therefore, it may 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" in the Control System Toolbox documentation.

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 "Frequency-Domain Validation of Linearization" on page 2-90.
- Simulate the nonlinear model and compare its time-domain response to that of the linearized model. For an example, see "Time-Domain Validation of Linearization" on page 2-86.

See Also

`linearize` | `slLinearizer`

Related Examples

- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

3

Frequency Response Estimation

- "Frequency Response Model Applications" on page 3-3
- "What Is a Frequency Response Model?" on page 3-4
- "Model Requirements" on page 3-6
- "Estimation Requires Input and Output Signals" on page 3-7
- "Creating Input Signals for Estimation" on page 3-9
- "Modifying Input Signals for Estimation" on page 3-24
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28
- "Estimate Frequency Response with Linearization-Based Input Using Linear Analysis Tool" on page 3-32
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Analyzing Estimated Frequency Response" on page 3-41
- "Troubleshooting Frequency Response Estimation" on page 3-50
- "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 3-63
- "Effects of Noise on Frequency Response Estimation" on page 3-73
- "Estimating Frequency Response Models with Noise Using Signal Processing Toolbox" on page 3-75
- "Estimating Frequency Response Models with Noise Using System Identification Toolbox" on page 3-77

- "Generate MATLAB Code for Repeated or Batch Frequency Response Estimation" on page 3-79
- "Managing Estimation Speed and Memory" on page 3-81

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 “Frequency-Domain Validation of Linearization” on page 2-90.

- Analyzing linear model dynamics.

  Designing controller for the plant represented by the estimated frequency response using Control System Toolbox software.

- Estimating parametric models.

  See “Estimating Frequency Response Models with Noise Using System Identification Toolbox” on page 3-77.

What Is a Frequency Response Model?

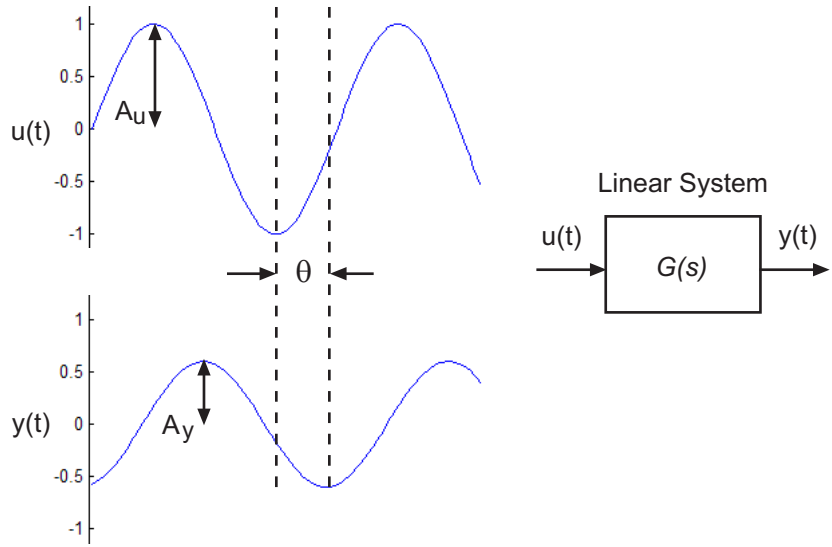
Frequency response describes the steady-state response of a system to sinusoidal inputs.

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:

$$G(s) = \frac{Y(s)}{U(s)}$$

$$|G(s)| = |G(j\omega)| = \frac{A_y}{A_u}$$

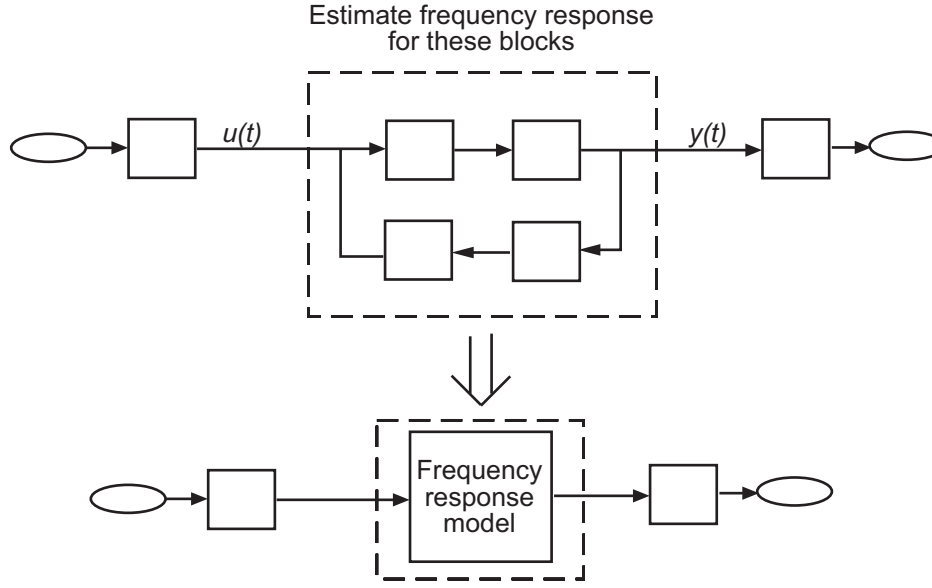
$$\theta = \angle \frac{Y(j\omega)}{X(j\omega)} = \tan^{-1}\left(\frac{\text{imaginary part of } G(j\omega)}{\text{real part of } G(j\omega)}\right)$$

where $Y(s)$ and $U(s)$ are the Laplace transforms of $y(t)$ and $u(t)$, respectively.

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).

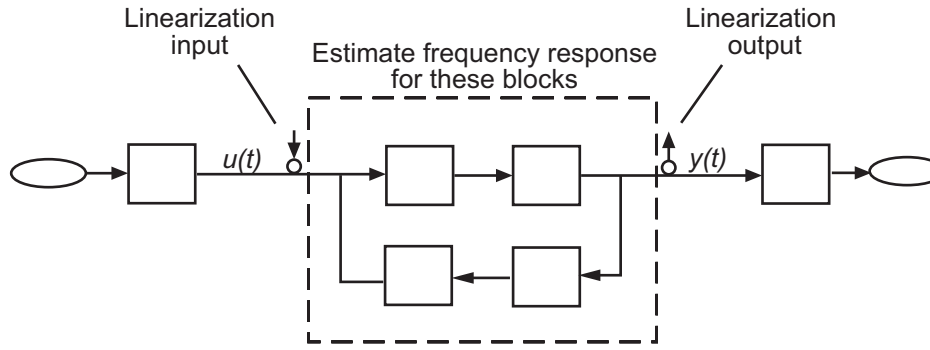
  For alternatives ways to model systems with noise, see "Estimating Frequency Response Models with Noise Using Signal Processing Toolbox" on page 3-75.

- 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 3-63.

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 "Creating Input Signals for Estimation" on page 3-9.



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.

Creating Input Signals for Estimation

In this section...
"Supported Input Signals" on page 3-9
"Creating Sinestream Input Signals" on page 3-9
"Creating Chirp Input Signals" on page 3-19

Supported Input Signals

Frequency response estimation uses 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. See "Creating Sinestream Input Signals" on page 3-9.
Chirp	• Your system is nearly linear in the simulation range. • You want to quickly obtain a response for a lot of frequency points. See "Creating Chirp Input Signals" on page 3-19.

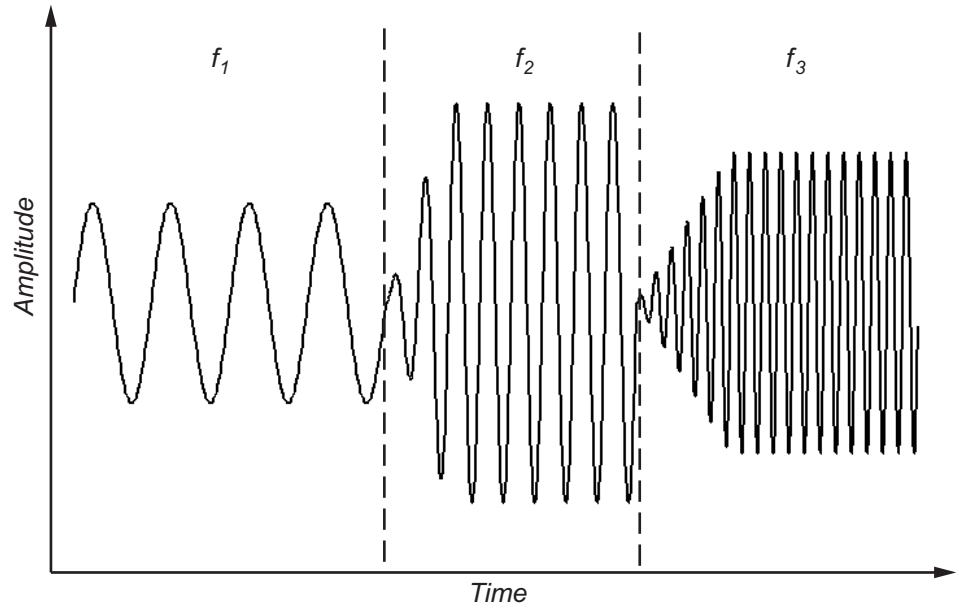
Creating Sinestream Input Signals

- "What Is a Sinestream Signal?" on page 3-10
- "Create Sinestream Signals Using Linear Analysis Tool" on page 3-10
- "Create Sinestream Signals (MATLAB Code)" on page 3-13

- "How Frequency Response Estimation Treats Sinestream Inputs" on page 3-15

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.



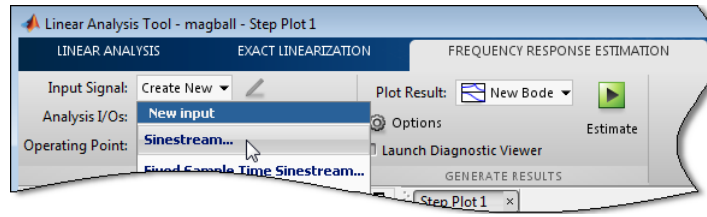
Create Sinestream Signals Using Linear Analysis Tool

This example shows how to create a sinestream input signal based upon a linearized model using the Linear Analysis Tool.

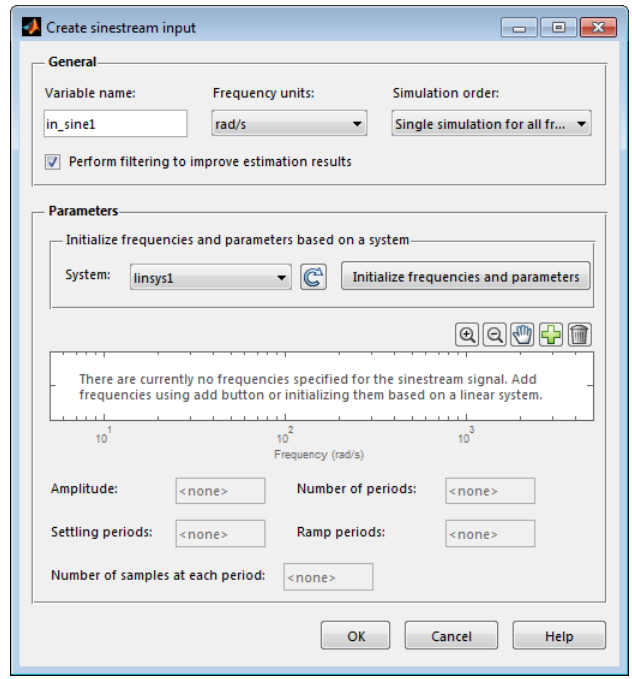
1 Obtain a linearized model, `linsys1`.

For example, see "Linearize Simulink Model" on page 2-43, which shows how to linearize a model.

2 In the Linear Analysis Tool, click the **Frequency Response Estimation** tab. In the **Input Signal** list, select **Sinestream**.

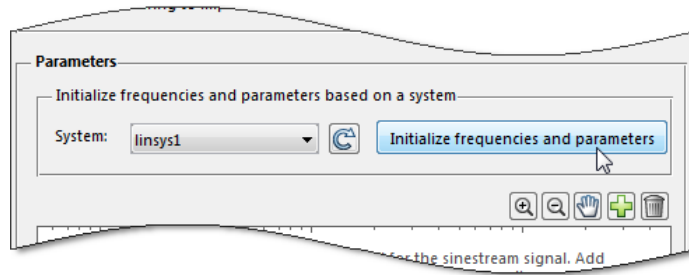


The Create sinestream input dialog box opens.

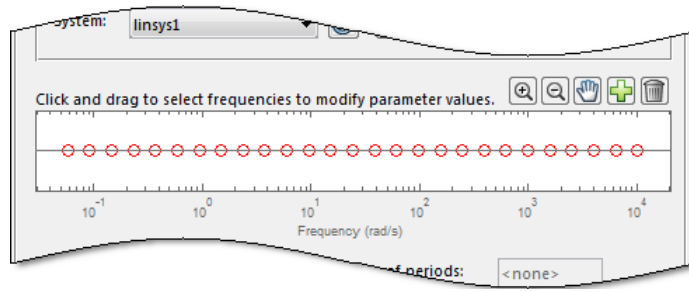


Note Selecting **Sinestream** creates a continuous-time signal. To generate a discrete-time signal, choose **Fixed Sample Time Sinestream** from the **Input Signal** list.

3 In the **System** list, select `linsys1`. Click **Initialize frequencies and parameters**.

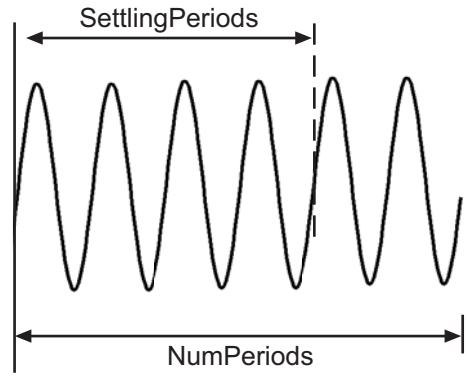


This action adds frequency points to the Frequency content viewer.

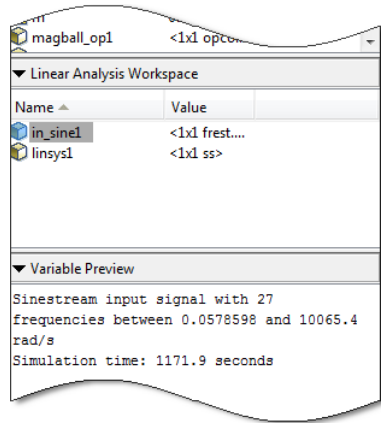


The software automatically selects frequency points based on the dynamics of `linsys1`. The software also automatically determines other parameters of the sinestream signal, including:

- 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 (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
```

For more information about sinestream options, see the `frest.Sinestream` reference page.

You can plot your input signal using `plot(input)`.

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'`

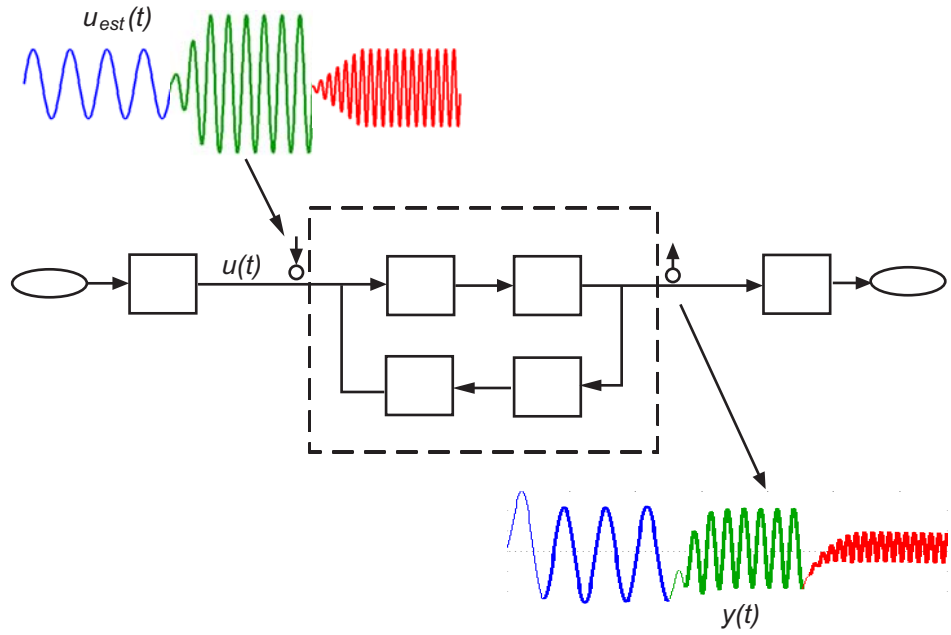
How Frequency Response Estimation Treats Sinestream Inputs

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.

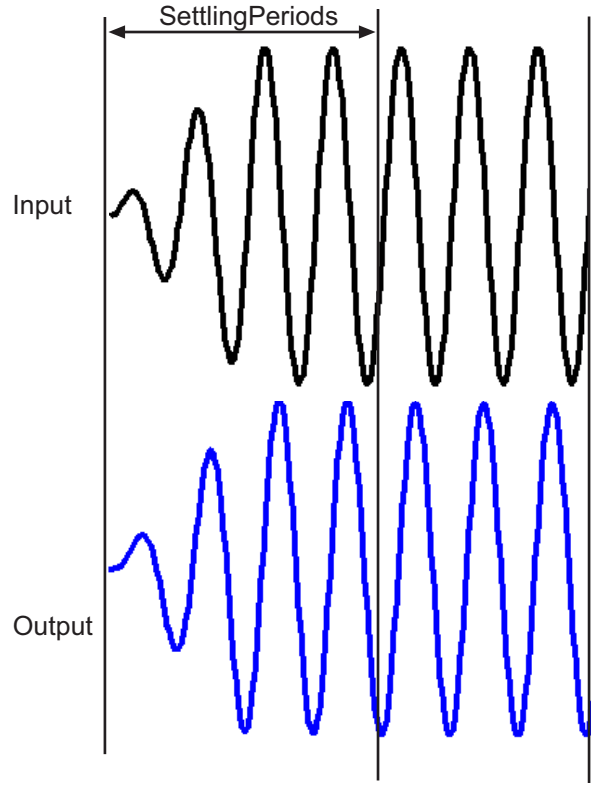
2 Simulates the output at the linearization output point.

`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.

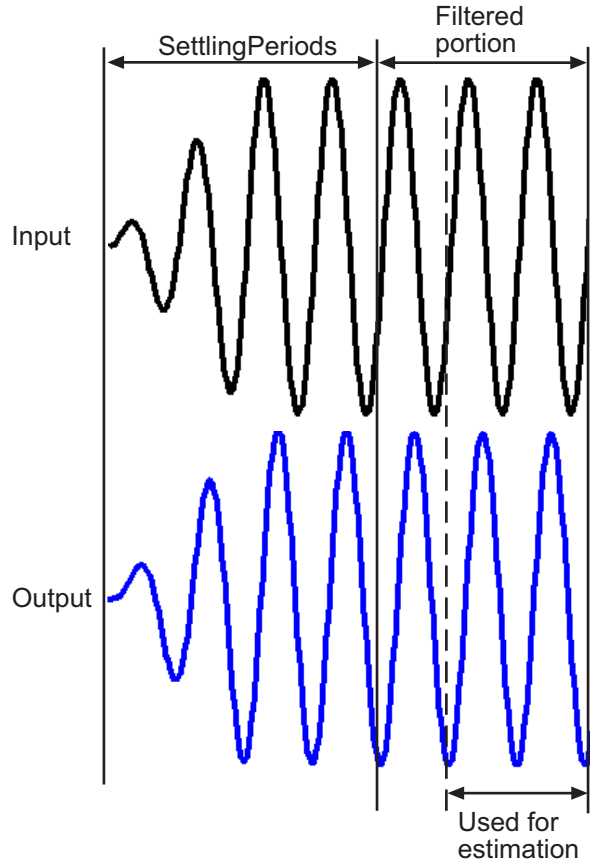


4 Filters the remaining portion of the output and the corresponding input signals at each input frequency using a bandpass filter.

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.

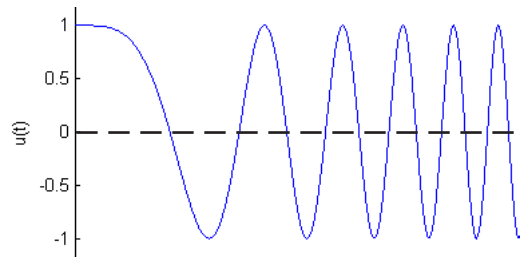
Creating Chirp Input Signals

- "What Is a Chirp Signal?" on page 3-19
- "Create Chirp Signals Using Linear Analysis Tool" on page 3-19
- "Create Chirp Signals (MATLAB Code)" on page 3-22

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, which excites the system at each frequency for several periods. See "Supported Input Signals" on page 3-9 for more information about choosing your signal.



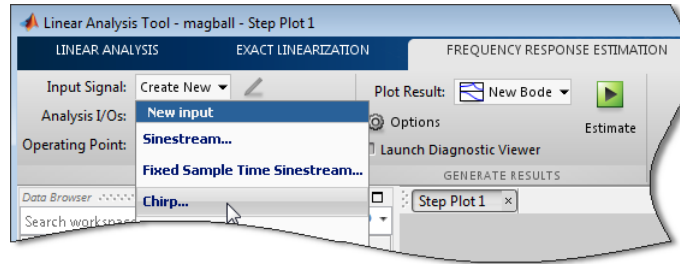
Create Chirp Signals Using Linear Analysis Tool

This example shows how to create a chirp input signal based upon a linearized model using the Linear Analysis Tool.

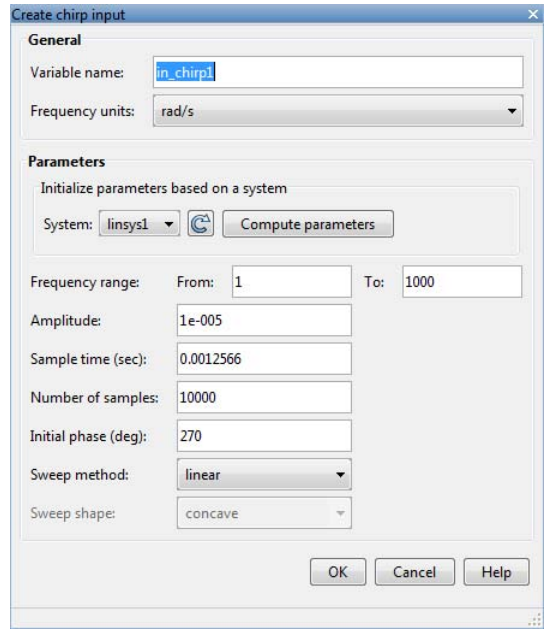
1 Obtain a linearized model, `linsys1`.

For example, see “Linearize Simulink Model” on page 2-43, which shows how to linearize a model.

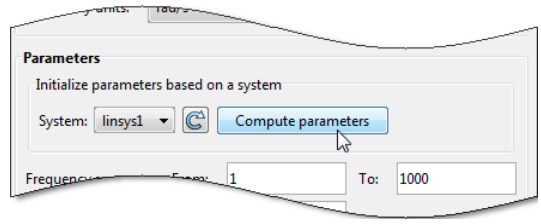
2 In the Linear Analysis Tool, click the **Frequency Response Estimation** tab. In the **Input Signal** list, select **Chirp**.



The Create chirp input dialog box opens.



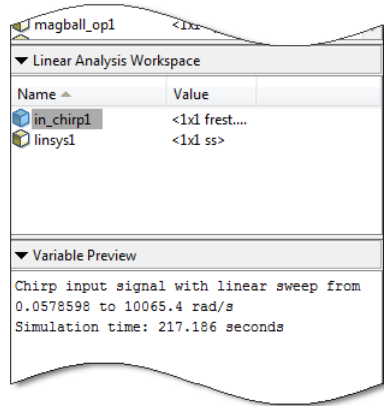
3 In the **System** list, select `linsys1` selected. Click **Compute parameters**.



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 (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)
```

For more information about chirp signal properties, see the `frest.Chirp` reference page.

You can plot your input signal using `plot(`*`input`*`)`.

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
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'`

Modifying Input Signals for Estimation

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 3-50.

Modifying Sinestream Input 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.

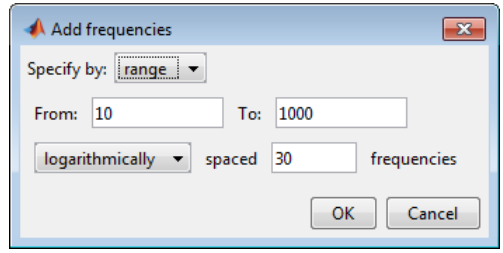
1. Create a sinestream input signal, `in_sine1`, as shown in “Create Sinestream Signals Using Linear Analysis Tool” on page 3-10.

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

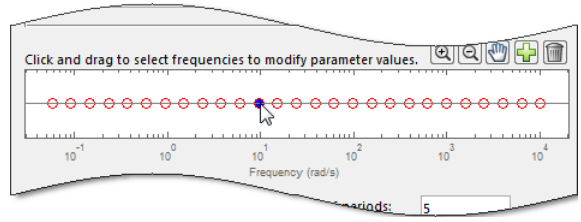
This example shows how to delete frequency points from an existing sinestream input signal using the Linear Analysis Tool.

1. Create a sinestream input signal, `in_sine1`, as shown in "Create Sinestream Signals Using Linear Analysis Tool" on page 3-10.

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

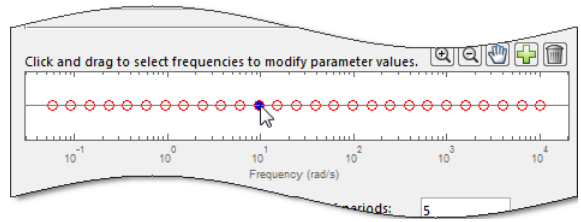
This example shows how to modify signal parameters of an existing sinestream input signal using the Linear Analysis Tool.

1. Create a sinestream input signal, `in_sine1`, as shown in "Create Sinestream Signals Using Linear Analysis Tool" on page 3-10.

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.

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.

Modifying Sinestream Input Signal (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.

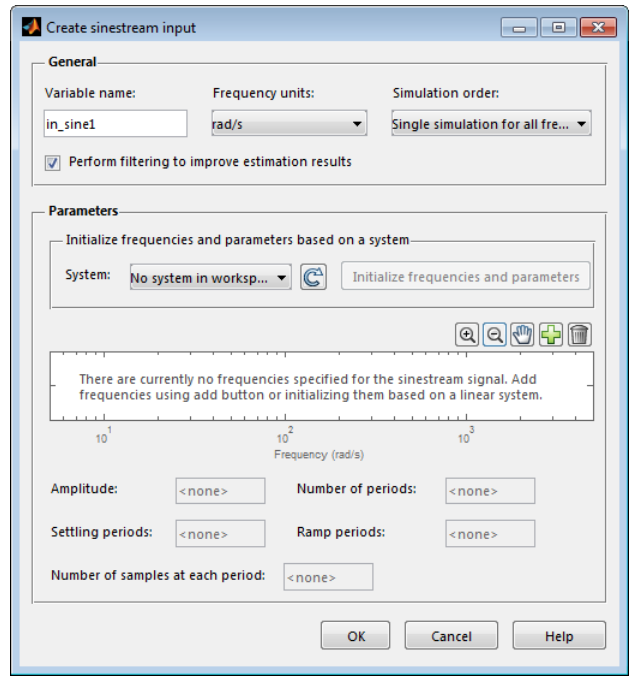
```
sys = 'scdDCMotor';
open_system(sys);
```

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 **Frequency Response Estimation** tab.

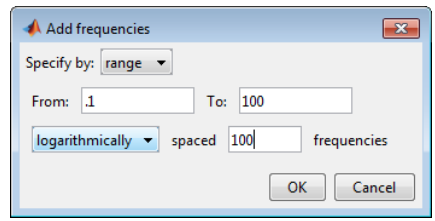
2. In the **Input Signal** list, select **Sinestream** to open the Create sinestream input dialog box.



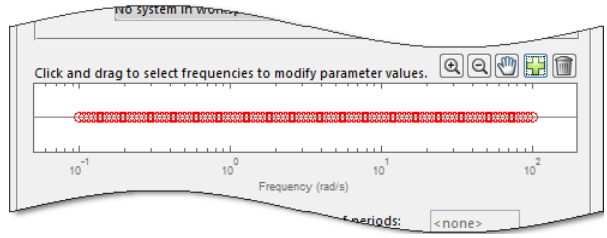
3 Click to open the Add frequencies dialog box. You can use this dialog box to add frequency points to the input signal.

4 Specify the frequency range for the input.

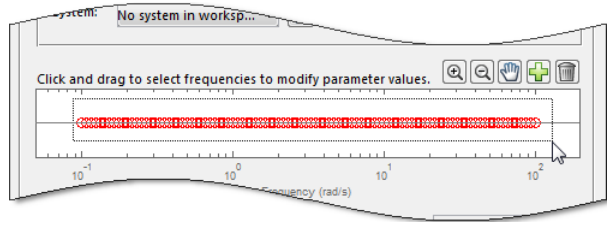
Enter .1 in the **From** box and 100 in the **To** box. Enter 100 in the box for the number of frequency points.



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.

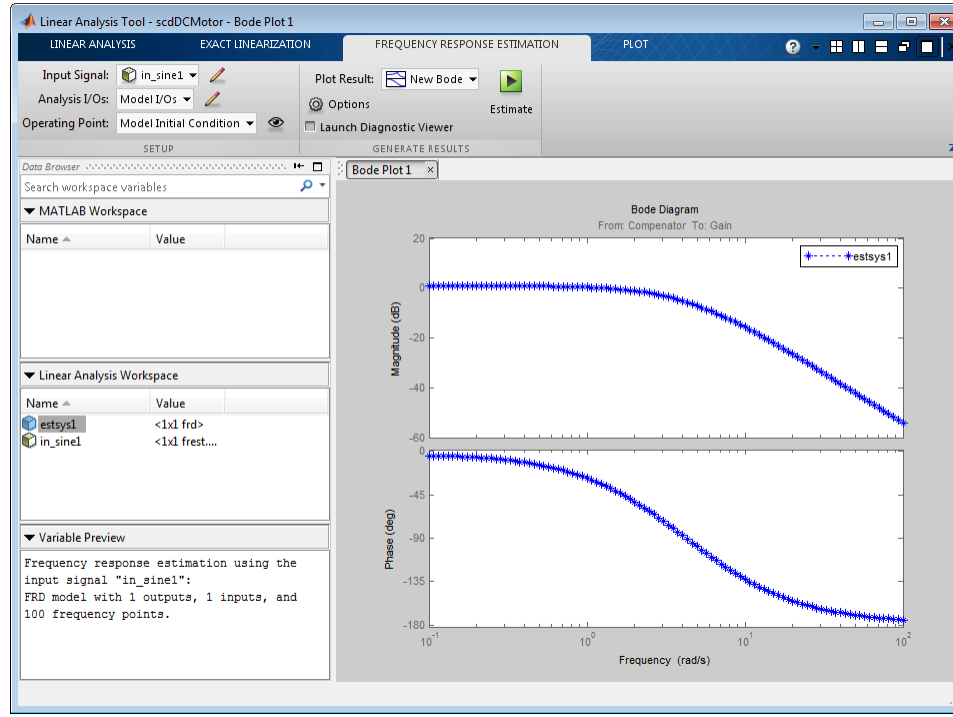
Enter 1 in the **Amplitude** box.

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 to estimate the frequency response. The frequency response estimation result, `estsys1`, appears in the **Linear Analysis Workspace**.


Linear Analysis Tool - scdDCMotor - Bode Plot 1
LINEAR ANALYSIS
EXACT LINEARIZATION
FREQUENCY RESPONSE ESTIMATION
PLOT
Input Signal:
in_sine1
Analysis I/Os:
Model I/Os
Operating Point:
Model Initial Condition
SETUP
Plot Result:
New Bode
Options
Launch Diagnostic Viewer
Estimate
GENERATE RESULTS
Data Browser
Search workspace variables
MATLAB Workspace
Name
Value
Linear Analysis Workspace
estsys1
<1x1 frd>
in_sine1
<1x1 frest....
Variable Preview
Frequency response estimation using the
input signal "in_sine1":
FRD model with 1 outputs, 1 inputs, and
100 frequency points.
Bode Plot 1
Bode Diagram
From: Compenator To: Gain
estsys1
Magnitude (dB)
Phase (deg)
Frequency (rad/s)

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.

Step 1. Linearize Simulink model.

- Open Simulink model.

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

- Open the Linear Analysis Tool for the model.

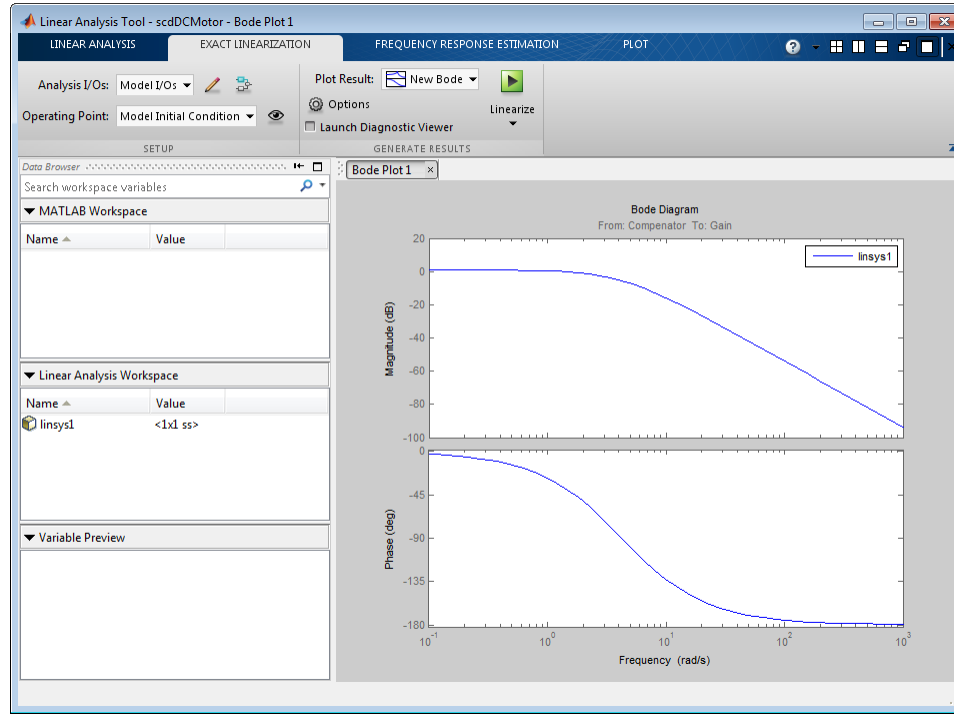
  In the Simulink Editor, select **Analysis > Control Design > Linear Analysis**.

- In the **Plot Result** list, choose **New Bode**.

- Linearize the model.

  Click . A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

  The software used the model initial conditions as the operating point to generate `linsys1`.



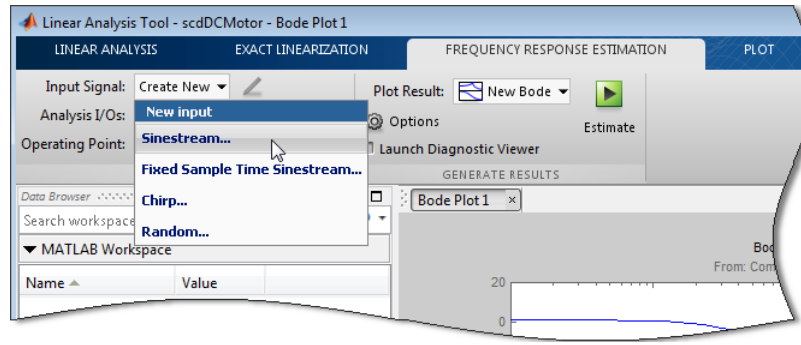
Step 2. Create sinestream input signal.

- Click the **Frequency Response Estimation** tab.

  In this tab, you estimate the frequency response of the model.

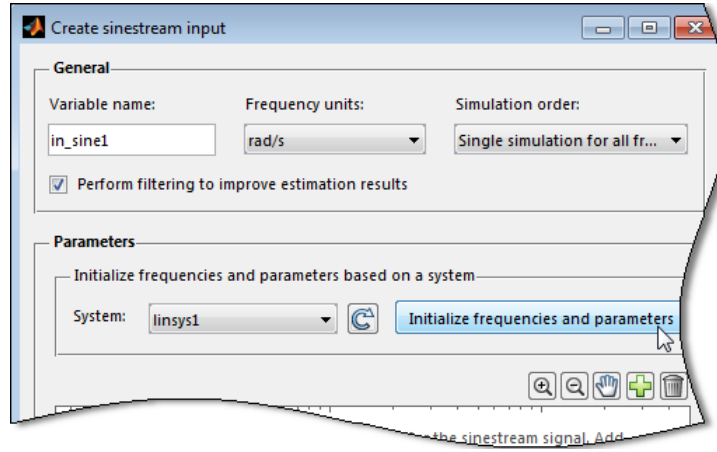
- Open the Create sinestream input dialog box.

  Select **Sinestream** from the **Input Signal** list to open the Create sinestream input dialog box.



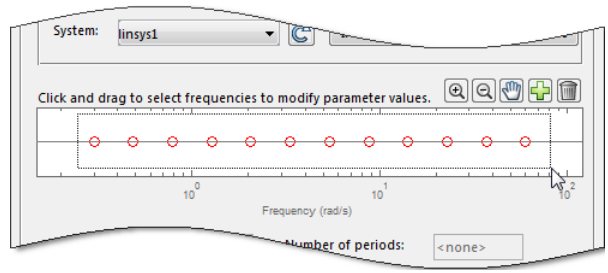
- Initialize the input signal frequencies and parameters based on `linsys1`.

  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`.

- In the Frequency content viewer of the Create sinestream input dialog box, select all the frequency points.



- Specify the amplitude of the input signal.

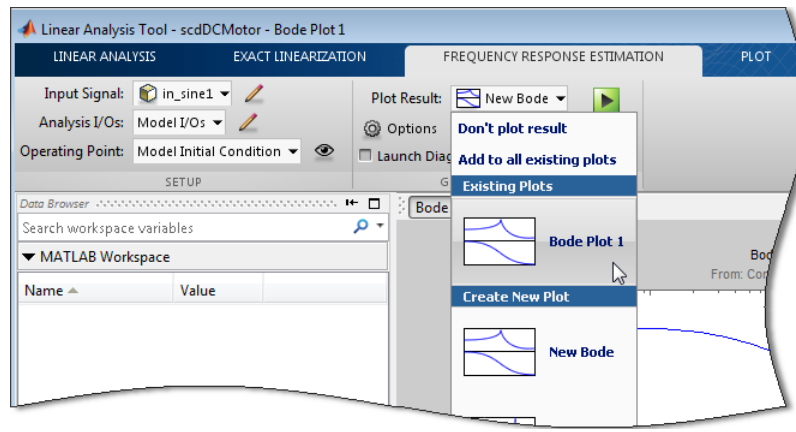
  Enter 1 in the **Amplitude** box.

- Create the input sinestream signal.

  Click **OK**. The input signal `in_sine1` appears in the **Linear Analysis Workspace**.

Step 3. Select the plot to display the estimation result.

In the **Plot Result** list, choose **Bode Plot 1** to add the next computed linear system to **Bode Plot 1**.

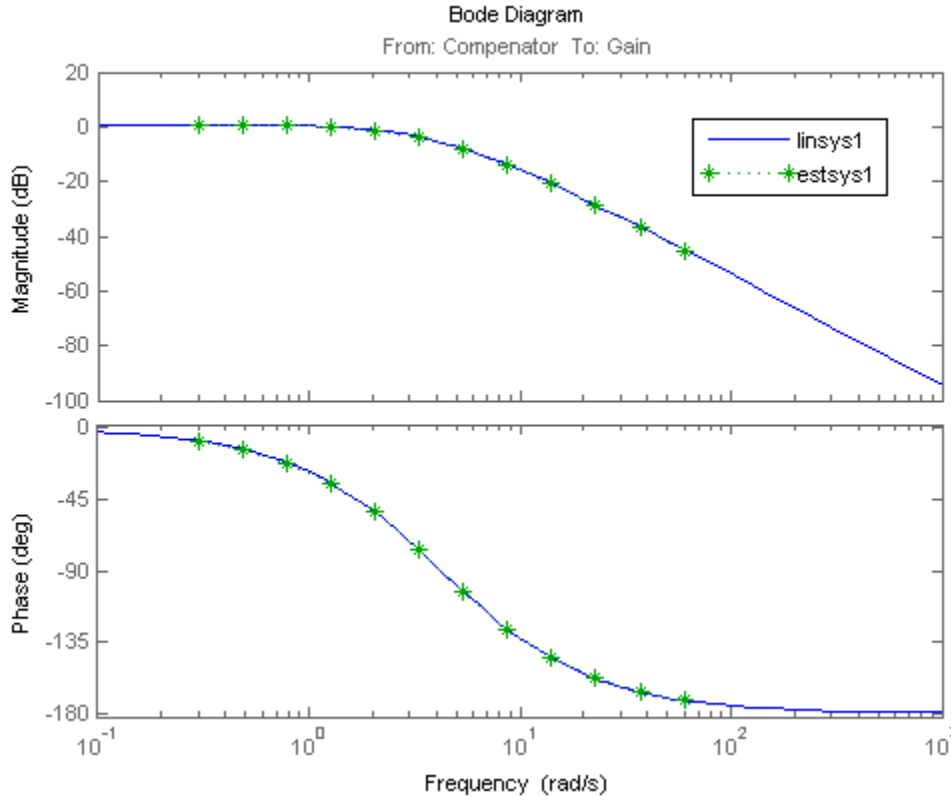


Step 4. Estimate frequency response.

Click . The estimated system, `estsys1`, appears in the **Linear Analysis Workspace**.

Step 5. 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.

Estimate Frequency Response (MATLAB Code)

Prerequisites

- Open Simulink model.

  Example:

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

  To learn more about general model requirements, see "Model Requirements" on page 3-6.

- Create an input signal for estimation.

  Example:

  ```
  io(1) = linio('scdplane/Sum1',1)
  io(2) = linio('scdplane/Gain5',1,'output')
  sys = linearize('scdplane',io);
  input = frest.Sinestream(sys)
  ```

  See "Creating Input Signals for Estimation" on page 3-9.

- (Optional) If your model has not reached steady state, initialize the model at a steady state operating point.

  You can check whether your model is at steady state by simulating the model. See `operspec` and `findop` reference pages.

1. Use linearization I/O points to specify input and output points for frequency response estimation.

   Example:

   ```
   io(1) = linio('scdplane/Sum1',1)
   io(2) = linio('scdplane/Gain5',1,'output')
   ```

   Caution Avoid placing I/O points on bus signals.

For more information about linearization I/O points, see "Specify Model Portion to Linearize" on page 2-13 and the `linio` reference page.

2. Identify all source blocks that generate time-varying signals in the signal path of the linearization outputs. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results.

   a. First, use `frest.findSources` to identify time-varying source blocks that can interfere with estimation. `frest.findSources` finds all time-varying source blocks in the signal path of the linearization output points.

      Example:

      Identify the time-varying source blocks in the `scdplane` model:

      ```
      srcblks = frest.findSources('scdplane',io);
      ```

   b. Next, to disable these blocks during estimation, use `frestimateOptions`.

      For example:

      ```
      opts = frestimateOptions;
      opts.BlocksToHoldConstant = srcblks;
      ```

   For more information, see the `frest.findSources` and `frestimateOptions` reference pages.

3. Estimate the frequency response.

   Example:

   ```
   [sysest,simout] = frestimate('scdplane',io,input,opts);
   ```

   `sysest` is the estimated frequency response. *`simout`* is the simulated output that is a `Simulink.Timeseries` object.

   For more information about syntax and argument descriptions, see the `frestimate` reference page.

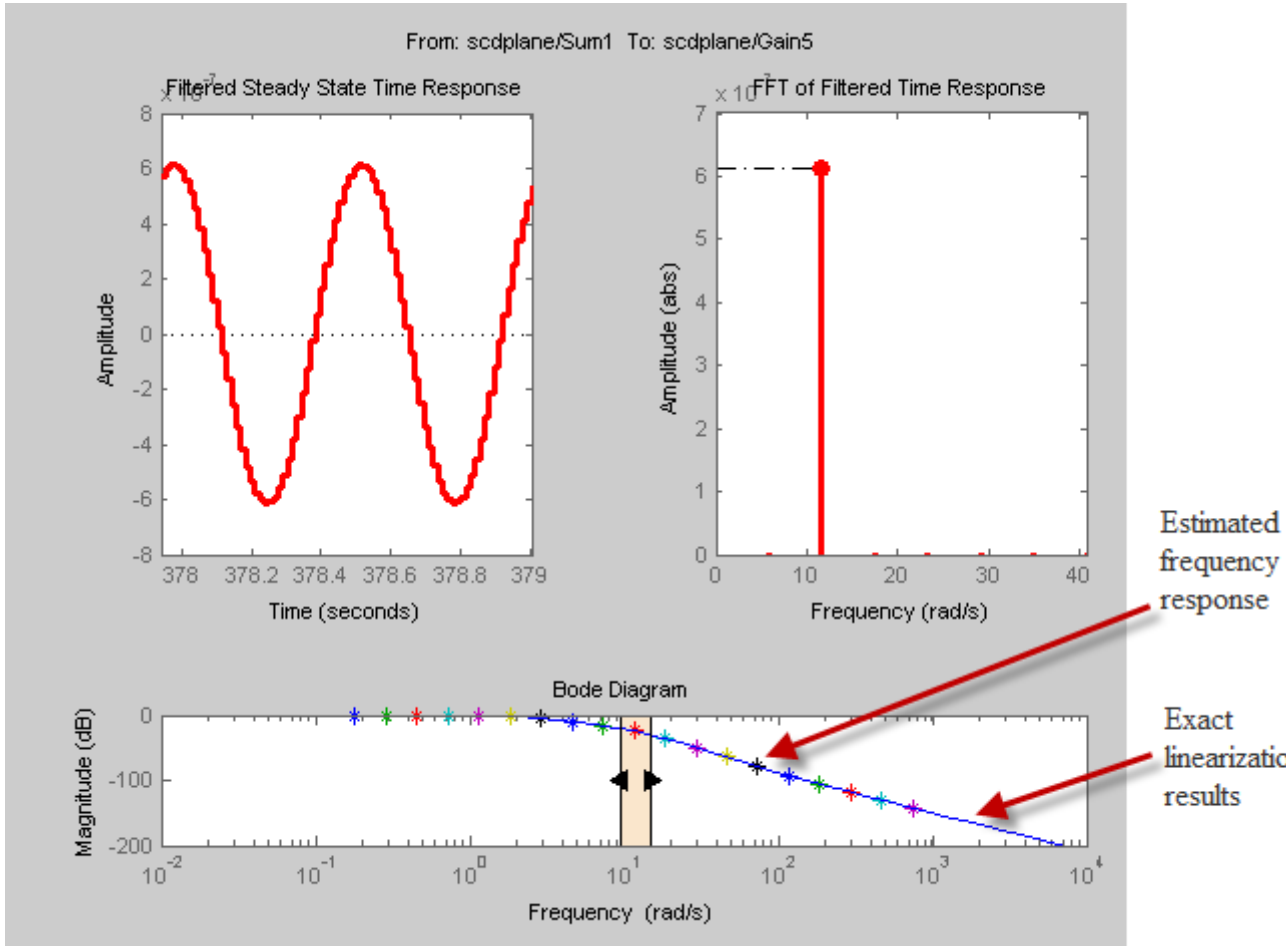
Tip To speed up your estimation or decrease memory requirements, see "Managing Estimation Speed and Memory" on page 3-81.

4 Open the Simulation Results Viewer to analyze the estimated frequency response. For example:

```
frest.simView(simout,input,sysest);
```

You can compare the estimated frequency response (*sysest*) to a system you linearized using exact linearization (*sys*):

```
frest.simView(simout,input,sysest,sys);
```



For more information, see “Analyzing Estimated Frequency Response” on page 3-41.

Analyzing Estimated Frequency Response

In this section...

"View Simulation Results" on page 3-41

"Interpret Frequency Response Estimation Results" on page 3-44

"Analyze Simulated Output and FFT at Specific Frequencies" on page 3-46

"Annotate Frequency Response Estimation Plots" on page 3-48

"Displaying Estimation Results for Multiple-Input Multiple-Output (MIMO) Systems" on page 3-49

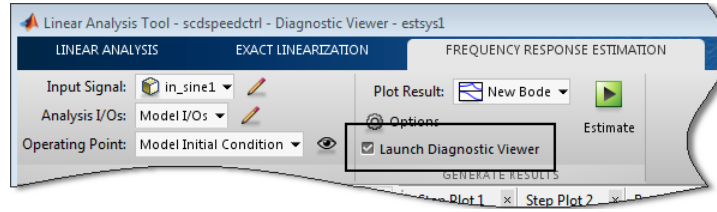
View Simulation Results

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 3-28.

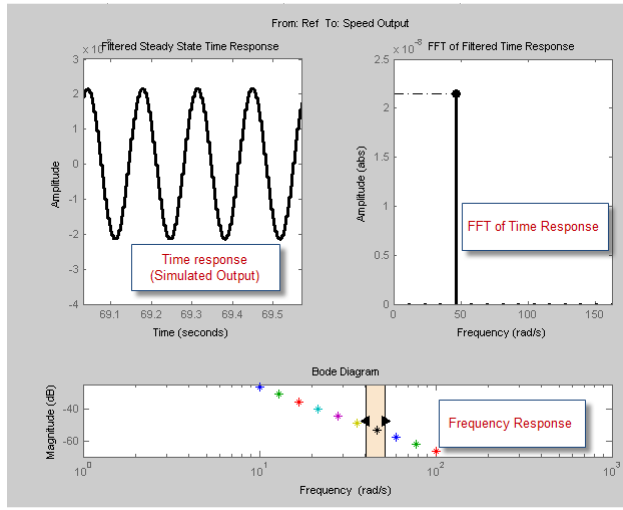
To open the Diagnostic Viewer after performing frequency response estimation in the Linear Analysis Tool:

1. In the **Frequency Response Estimation** tab, before performing the estimation task, select the **Launch Diagnostic Viewer** check box.



   This action sets the Diagnostic Viewer to open when the frequency response estimation is performed.

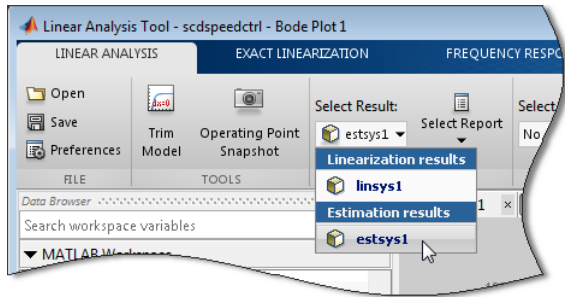
2. Click to estimate the frequency response of the model.



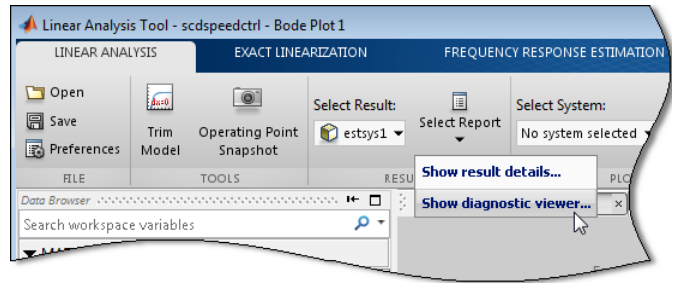
The Diagnostic Viewer appears in the plot pane.

To open the Diagnostic Viewer to view an estimated model in the Linear Analysis Tool:

1. In the **Linear Analysis Tab**, select the estimated model using the **Select Result** list.



2. Select **Show diagnostic viewer** in the **Show Report** list.



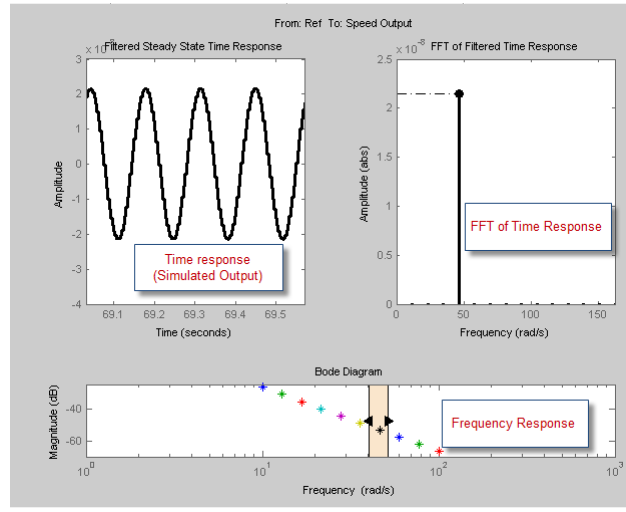
View Simulation Results (MATLAB Code)

Use the Simulation Results Viewer to analyze the results of your frequency response estimation, obtained by performing the steps in "Estimate Frequency Response (MATLAB Code)" on page 3-37.

Open the Simulation Results Viewer using:

```
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

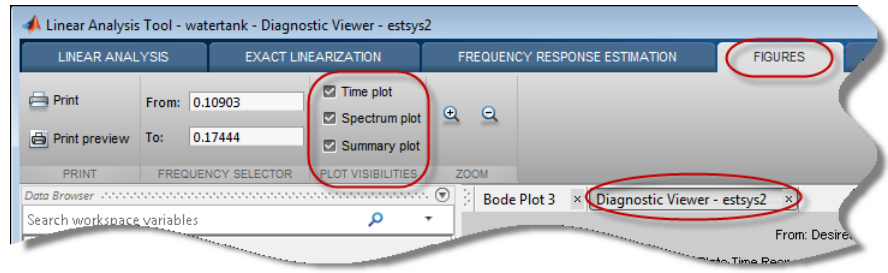
By default, the Simulation Results Viewer shows these plots:

- "Select Plots Displayed in Simulation Results Viewer" on page 3-44
- "Frequency Response" on page 3-44
- "Time Response (Simulated Output)" on page 3-45
- "FFT of Time Response" on page 3-45

Select Plots Displayed in Simulation Results Viewer

- To select the plots displayed in the Diagnostic Viewer using the Linear Analysis Tool:

  1. In the **Figures** tab, select the Diagnostic Viewer plot.
  2. 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.

- 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 3-50 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.

You can usually improve estimation results by either modifying your input signal 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 3-50 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 3-50.

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 estimation results. For more information about filtering during estimation, see "How Frequency Response Estimation Treats Sinestream Inputs" on page 3-15.

FFT of Time Response

Use this plot to analyze the spectrum of the simulated output.

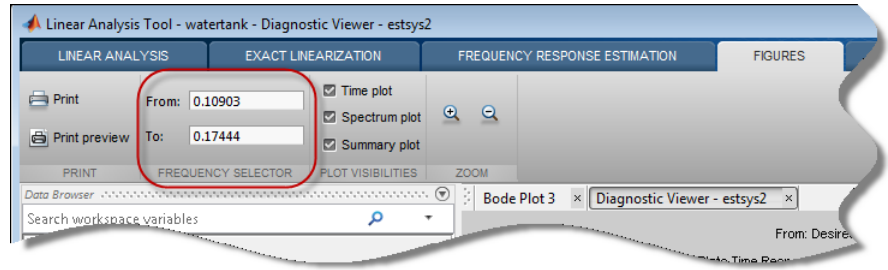
For example, you can use the spectrum to identify strong nonlinearities. When the FFT plot shows large amplitudes at frequencies other than the input signal, your model is operating outside of linear range. If you are interested in analyzing the linear response of your system for small perturbations, explore possible solutions in "FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency" on page 3-54.

Analyze Simulated Output and FFT at Specific Frequencies

Using the Diagnostic Viewer in Linear Analysis Tool

To analyze the results in the Diagnostic Viewer, use the controls in the **Figures** tab of the Linear Analysis Tool.

1. In the **Figures** tab, select the Diagnostic Viewer plot.
2. 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.

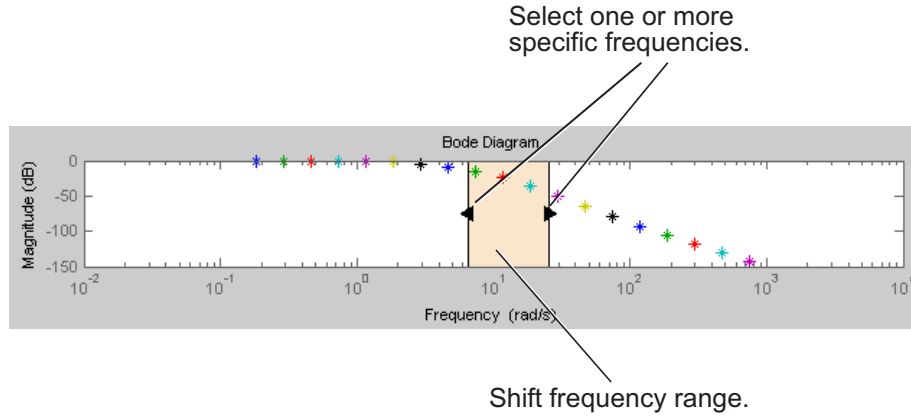


Using the Simulation Results Viewer

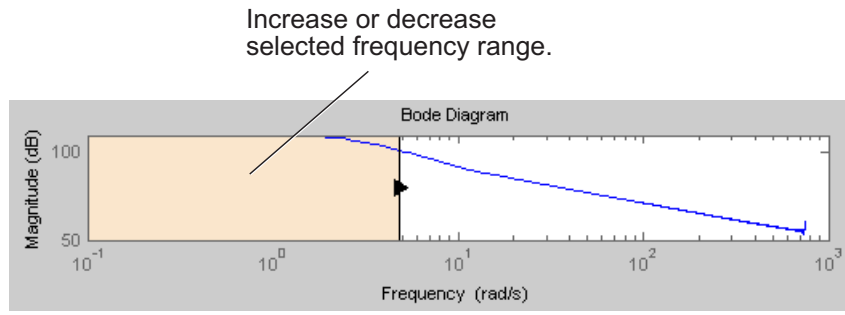
In the Simulation Results Viewer, use Bode controls to display the simulated output and its spectrum at specific frequencies.

If you used the sinestream input signal in the estimation:

- Drag 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.

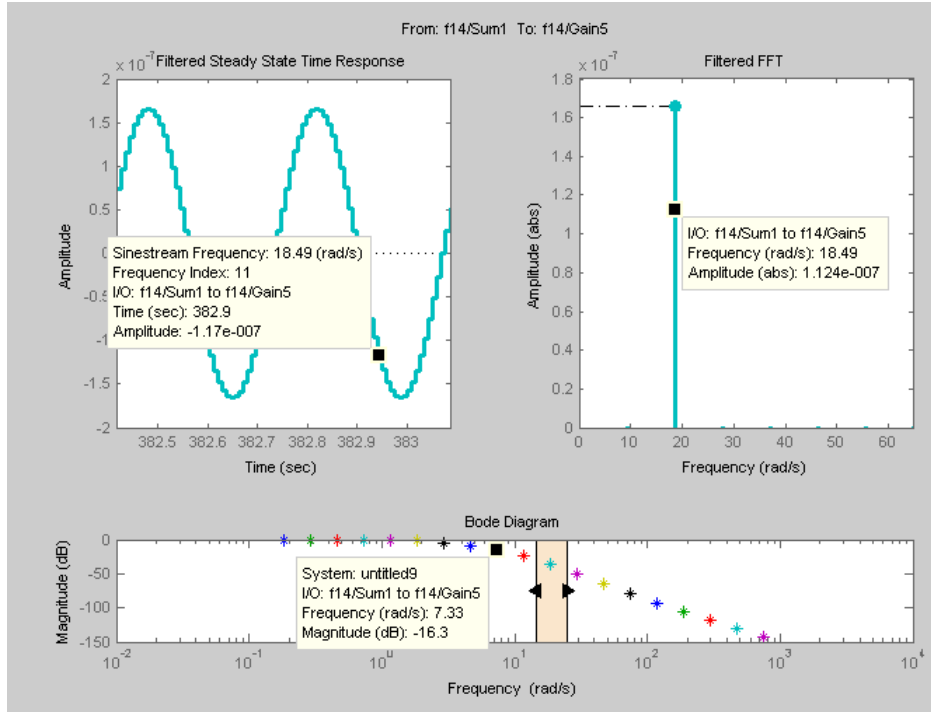


If you used the chirp input signal in the estimation, drag the shaded region to increase or decrease the frequency range of the displayed time response and FFT.



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;
```

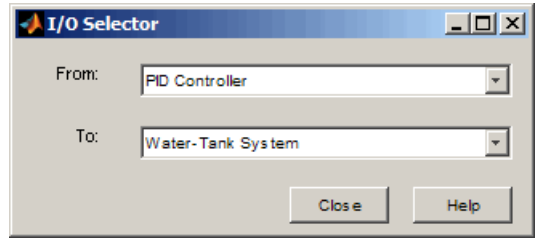
```
input.SettlingPeriods(11) = 75;
```

To modify the sinestream in the Linear Analysis Tool, see “Modifying Sinestream Input Signal Using Linear Analysis Tool” on page 3-24

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 the Simulation Results Viewer, right-click any plot, and select **I/O Selector**.
2. Choose the input channel in the **From** list. Choose the output channel in the **To** list.



Troubleshooting Frequency Response Estimation

In this section...
"When to Troubleshoot" on page 3-50
"Time Response Not at Steady State" on page 3-50
"FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency" on page 3-54
"Time Response Grows Without Bound" on page 3-56
"Time Response Is Discontinuous or Zero" on page 3-58
"Time Response Is Noisy" on page 3-60

When to Troubleshoot

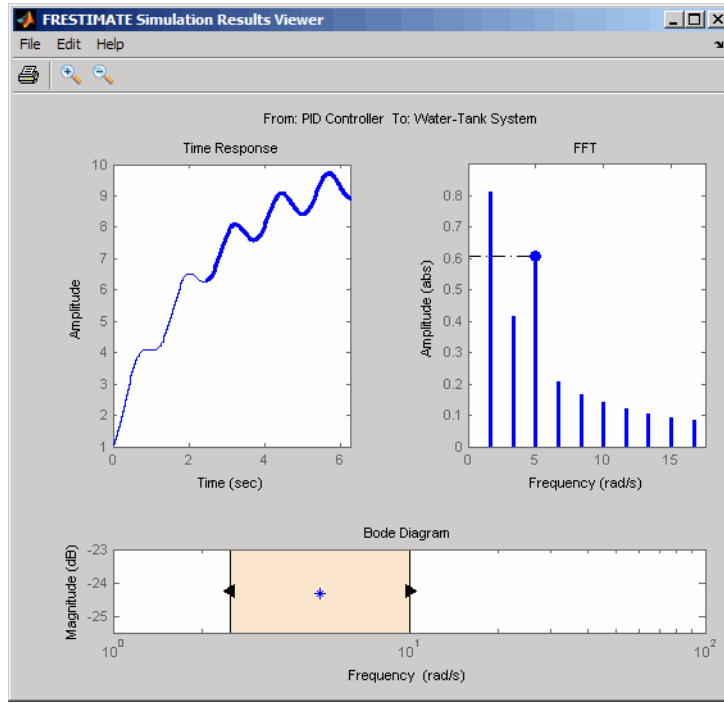
After you estimate the frequency response, you can analyze the results. If the frequency response plot does not match the expected behavior of your system, you can use the time response and FFT plots to help you improve the results.

If your estimation is slow or you run out of memory during estimation, see "Managing Estimation Speed and Memory" on page 3-81.

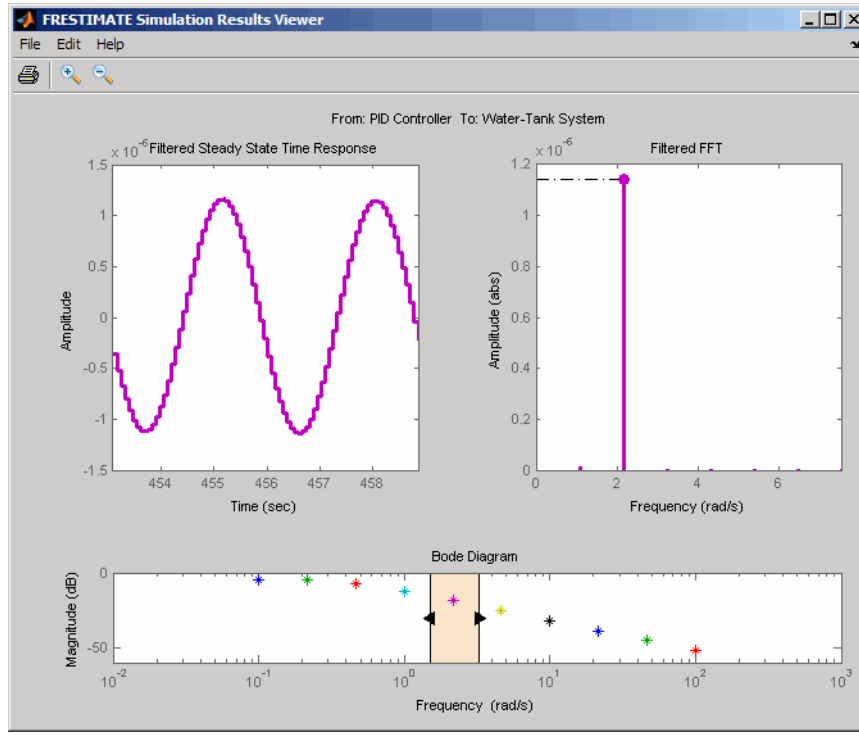
Time Response Not at Steady State

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 "Modifying Input Signals for Estimation" on page 3-24. • 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 3-63.
(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 "Modifying Input Signals for Estimation" on page 3-24. • 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 "Modifying Input Signals for Estimation" on page 3-24.

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 "Modifying Input Signals for Estimation" on page 3-24.

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`. For example:

      ```
      sysest = fdel(sysest,input2.Frequency)
      ```

   b. Concatenate the original and recomputed responses using `fcat`. For example:

      ```
      sys_combined = fcat(sysest2,sysest)
      ```

You can analyze the recomputed frequency response, as described in "Analyzing Estimated Frequency Response" on page 3-41.

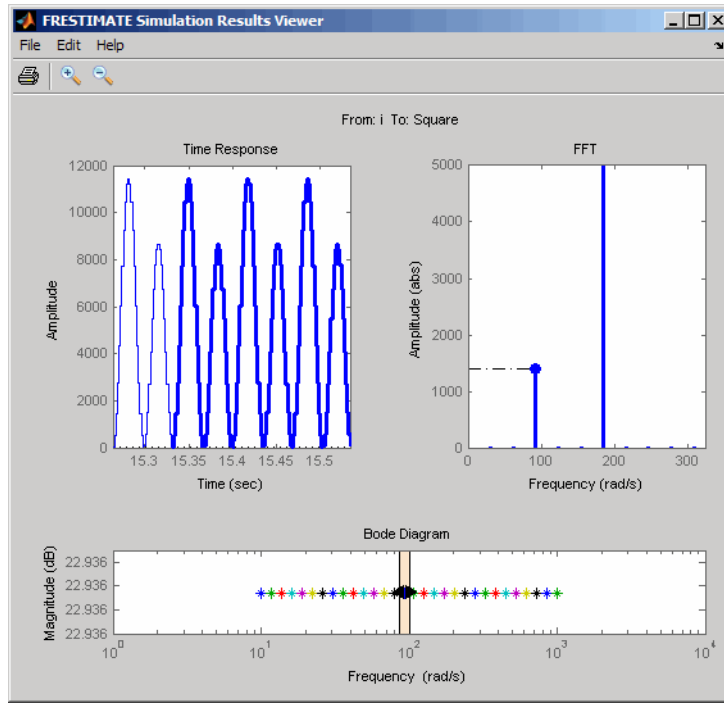
For an example of frequency response estimation with time-varying source blocks, see "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 3-63

FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency

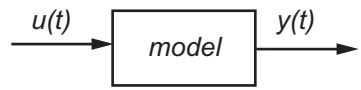
What Does This Mean?

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 causes problems when you want to analyze your linear system response 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.

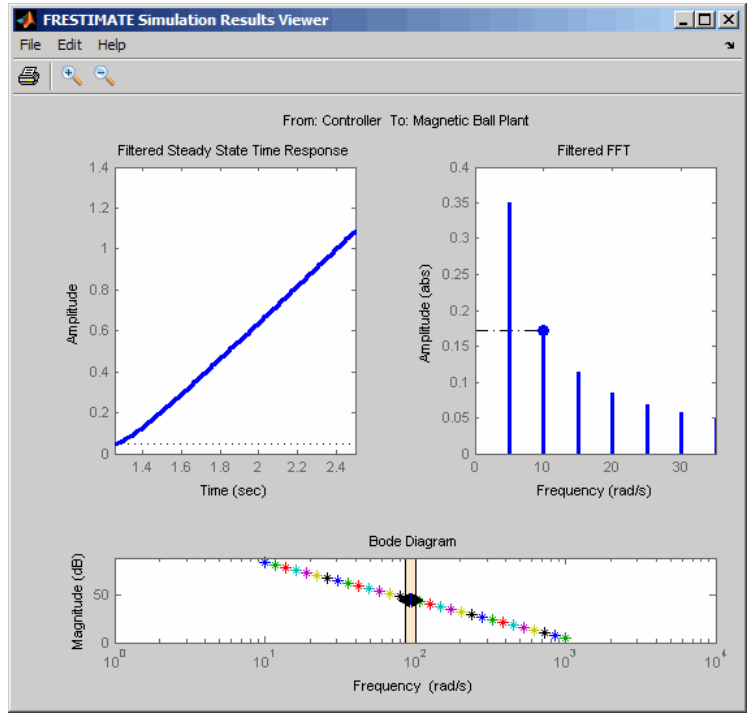
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- "Modifying Input Signals for Estimation" on page 3-24

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 at Model Operating Point" on page 2-43 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 "Steady-State Operating

Possible Cause	Action
	Points (Trimming) from Specifications" on page 1-14.
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.

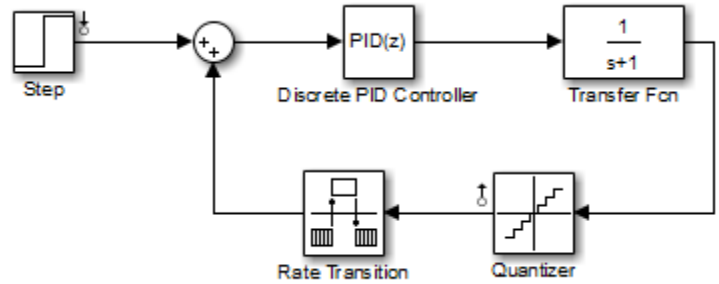
Time Response Is Discontinuous or Zero

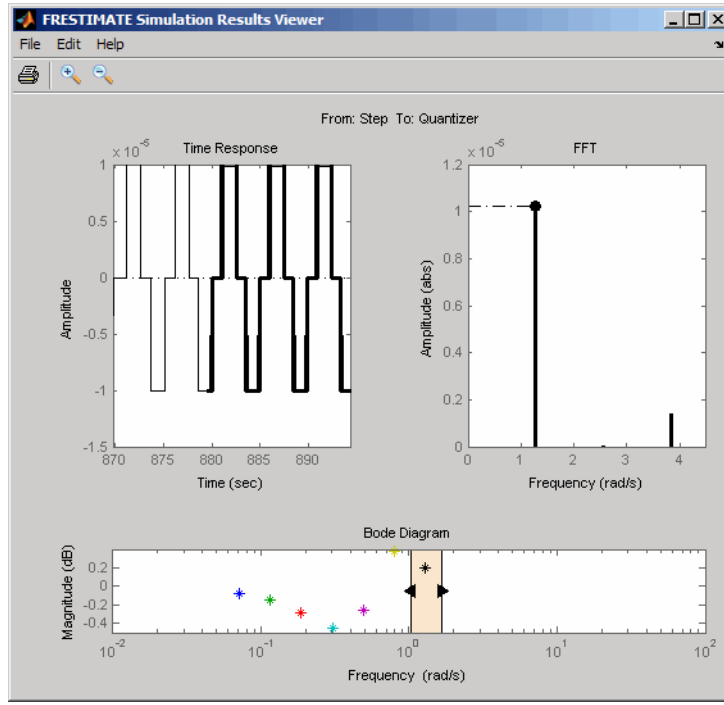
What Does This Mean?

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.

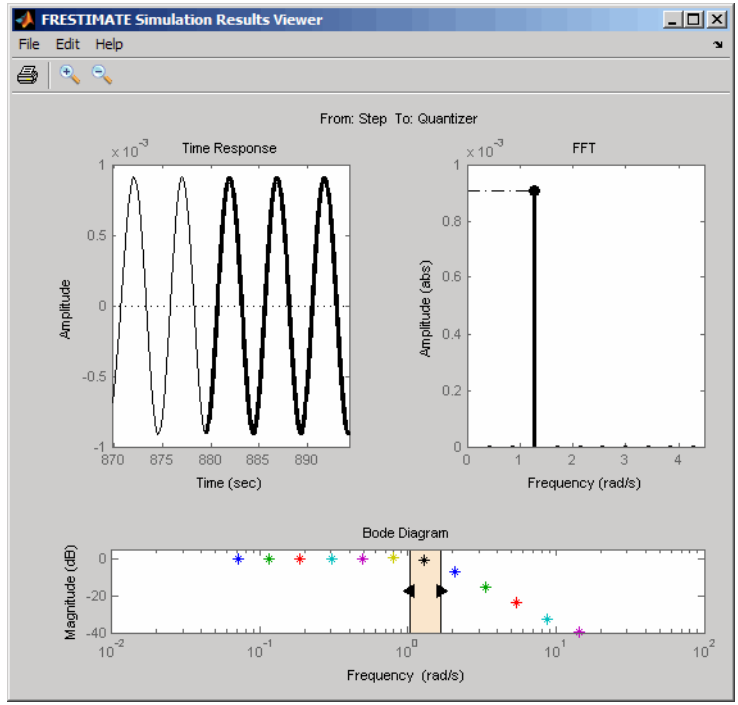




How Do I Fix It?

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.



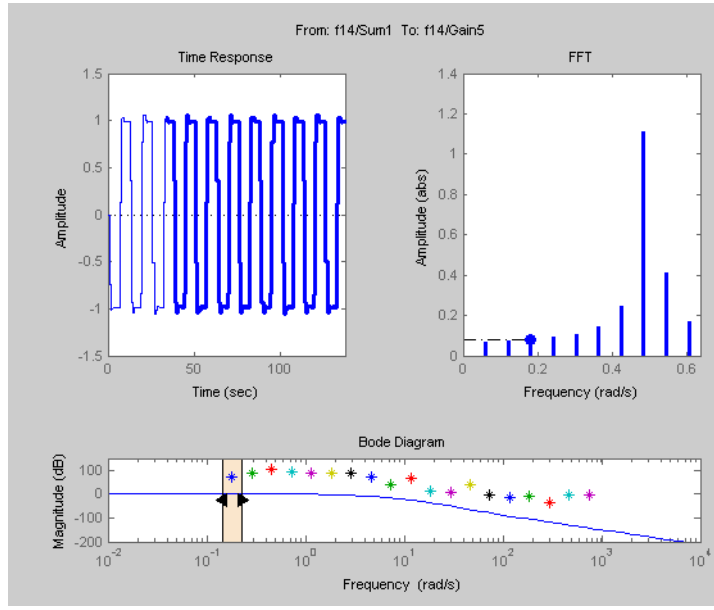
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- “Modifying Input Signals for Estimation” on page 3-24

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 "Estimating Frequency Response Models with Noise Using Signal Processing Toolbox" on page 3-75 and "Estimating Frequency Response Models with Noise Using System Identification Toolbox" on page 3-77.

For a related example, see "Effects of Noise on Frequency Response Estimation" on page 3-73.

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. Select the visualization for the linearized model.

      In the **Plot Result** list, select **New Bode**.

   d. Click .

      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 **Frequency Response Estimation** tab, select **Sinestream** in the **Input Signal** list.

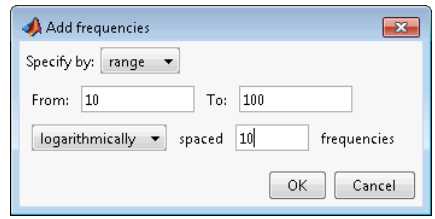
   b. Open the Add frequencies dialog box.

Click .

c Specify the input sinestream frequency range and number of frequency points.

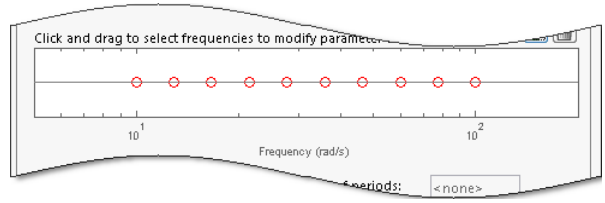
Enter 100 in the **To** box.

Enter 10 in the box for the number of frequency points.

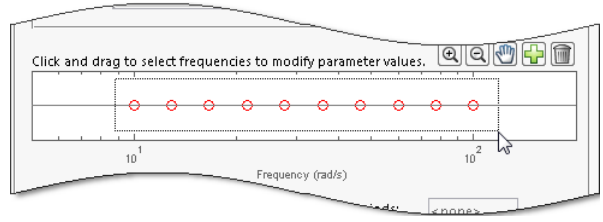


Click **OK**.

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.

For this example, 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 Select the visualization for the estimated model.

In the **Plot Result** list of the Linear Analysis Tool, select **Bode Plot 1**.

5 Set the Diagnostic Viewer to open when estimation is performed.

Select the **Launch Diagnostic Viewer** check box.

6 Estimate the frequency response for the model.

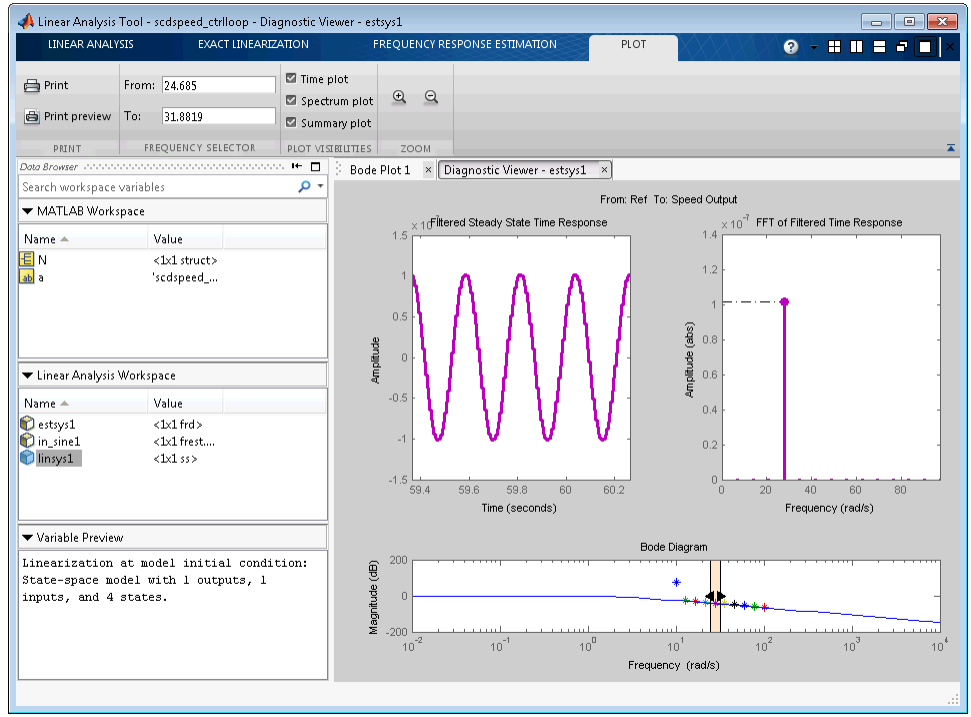
Click . The frequency response estimation result, `estsys1`, appears in the **Linear Analysis Workspace**.

7 Compare the estimated model and the linearized model.

a Click the **Diagnostic Viewer - estsys1** tab in the plot pane of the Linear Analysis Tool.

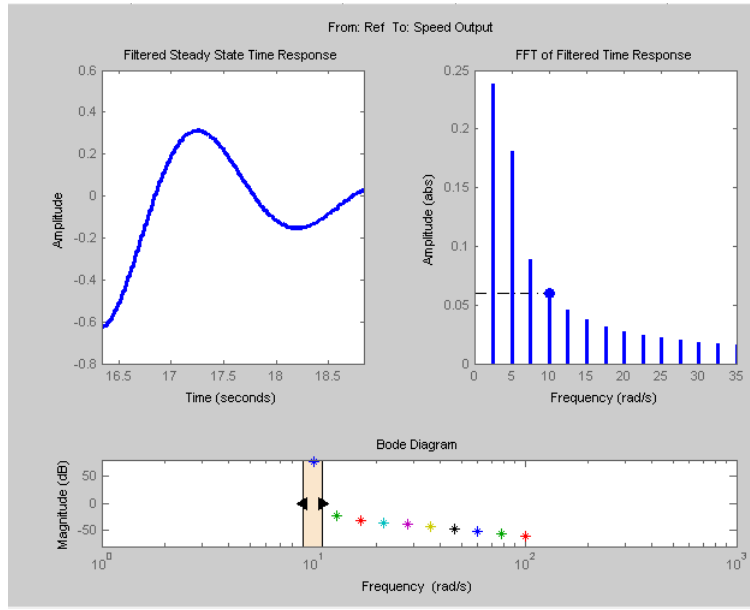
b Click and drag `linsys1` onto the Diagnostic Viewer to add `linsys1` to the Diagnostic Viewer plots.

c Click the **Figures** tab of the Linear Analysis Tool.



d Configure the Diagnostic Viewer to show only the frequency point where the estimation and linearization results do not match.

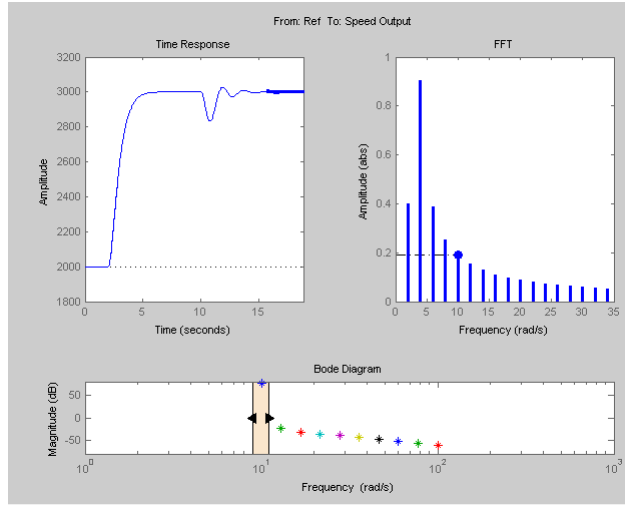
In the **Frequency Selector** section, enter 9 in the **From** box and 11 in the **To** box to set the frequency range that is analyzed in the Diagnostic Viewer.



The **Filtered Steady State Time Response** plot depicts 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.

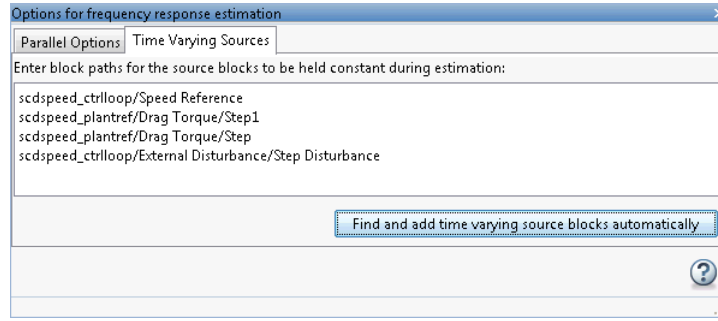
8 Find and disable the time-varying sources within the model.

a Open the Options for frequency response estimation dialog box.

On the **Frequency Response Estimation** tab, in the **Generate Results** section, click **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.



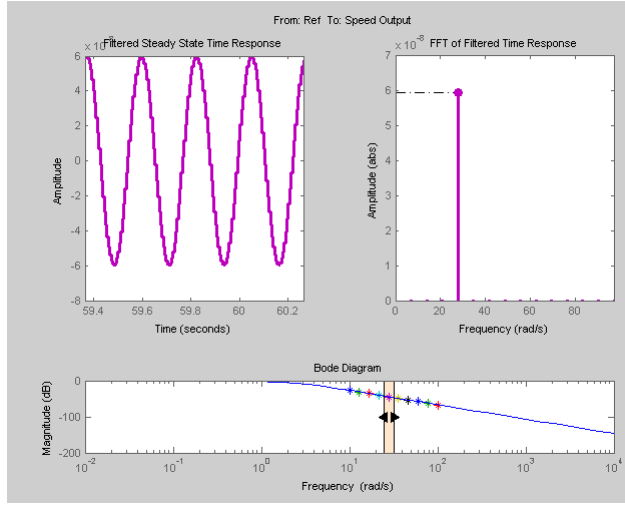
9 Estimate the frequency response for the model.

Click . The frequency response estimation result for the model, `estsys2`, appears in the **Linear Analysis Workspace**.

10 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 to add `linsys1` to the Diagnostic Viewer plots.



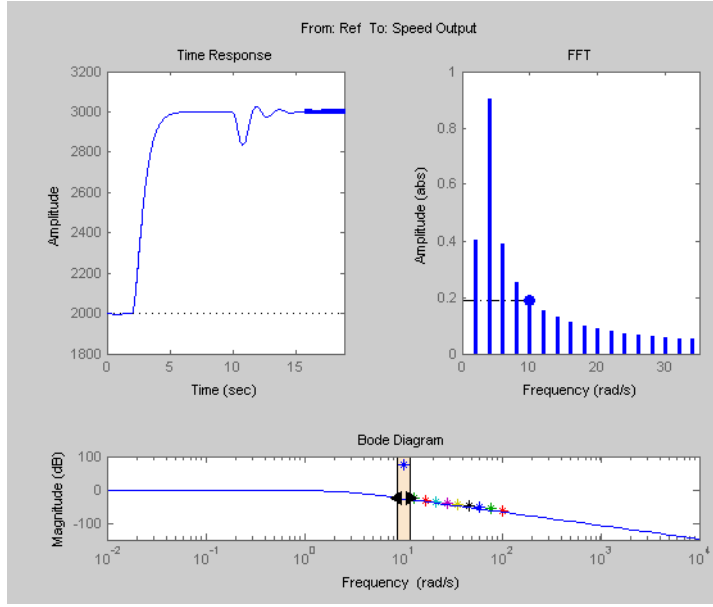
The frequency response obtained by holding the time-varying sources constant matches the exact linearization results.

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);
```

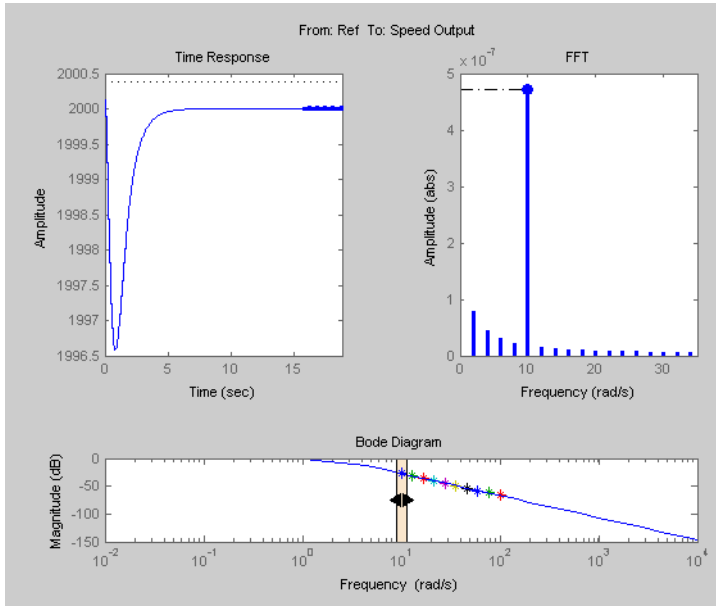
Create a `frestimate` options set to disable the blocks.

```
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**.

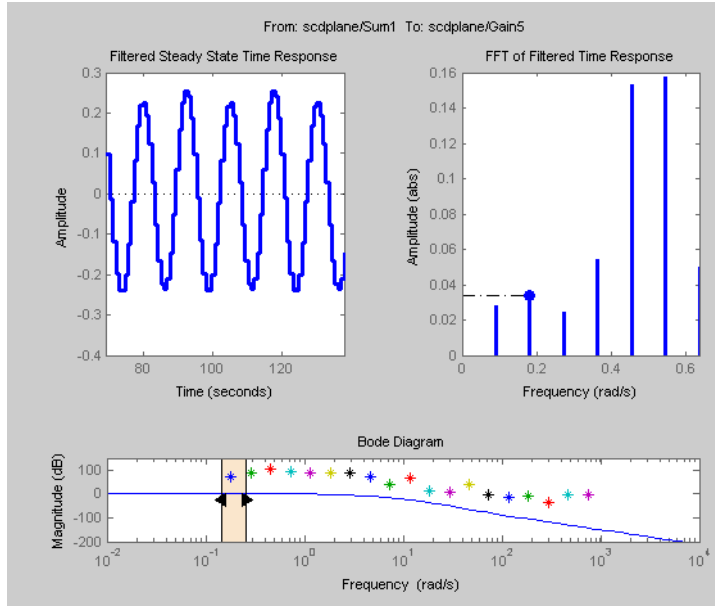


Effects of Noise on Frequency Response Estimation

Compare the linear model obtained using exact linearization techniques with the estimated frequency response:

```
mdl = 'scdplane';
open_system(mdl)
io(1) = linio('scdplane/Sum1',1)
io(2) = linio('scdplane/Gain5',1,'output')
sys = linearize(mdl,io);
in = frest.Sinestream(sys);
[sysest,simout] = frestimate(mdl,io,in);
frest.simView(simout,in,sysest,sys)
```

The resulting frequency response does not match the linearization results due to the effects of the Pilot and Wind Gust Disturbance blocks. To view the effects of effects of the noise on the time response of the first frequency, right-click the time response plot and make sure **Show filtered steady state output only** is selected.



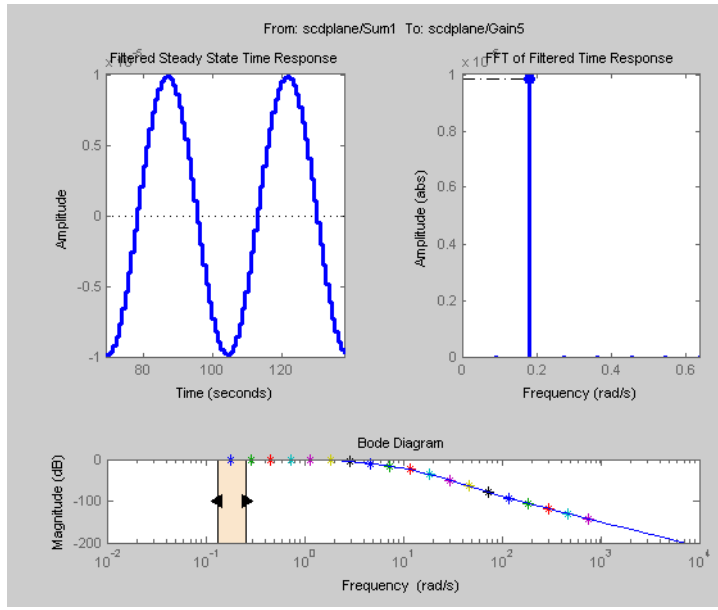
Locate the source blocks:

```
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);
```

The resulting frequency response matches the exact linearization results.



Estimating 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'),...
                      linoptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

Simulate the model to obtain the output at the linearization output point:

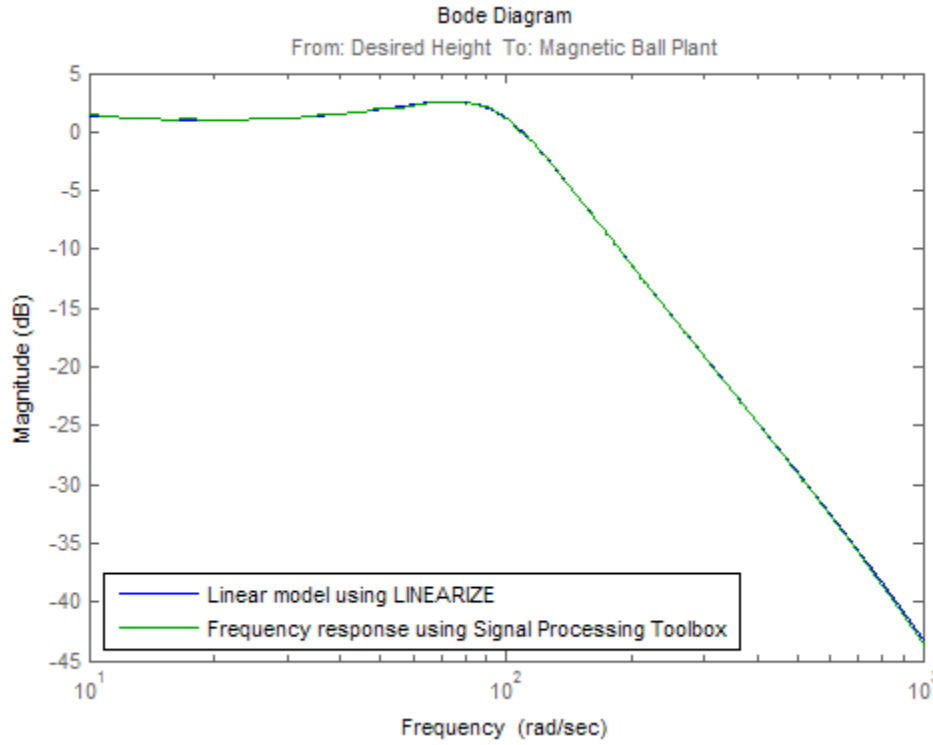
```
[sysest,simout] = frestimate('magball',io,in,op);
```

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.

Estimating 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:

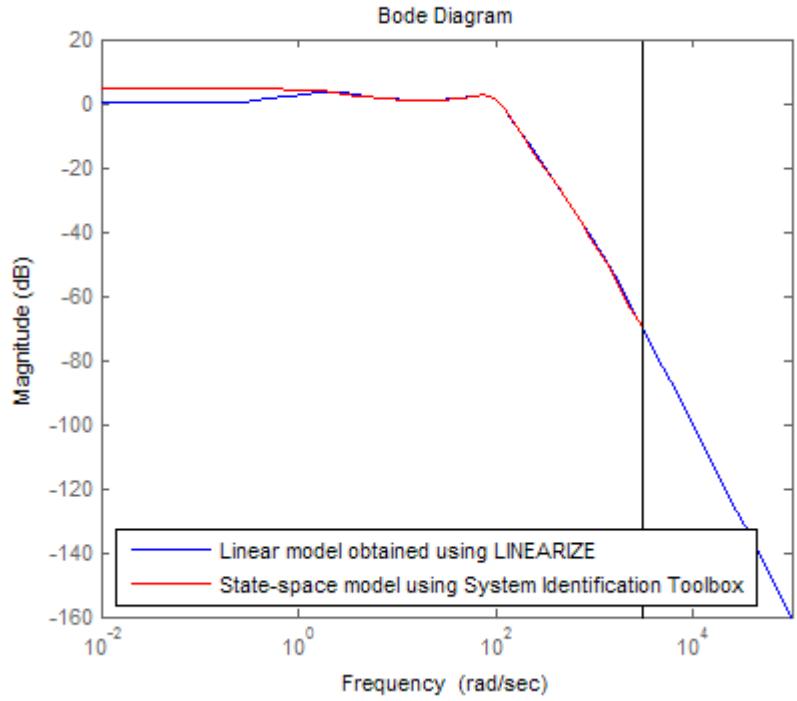
```
op = findop('magball',operspec('magball'),...
                      linoptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

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

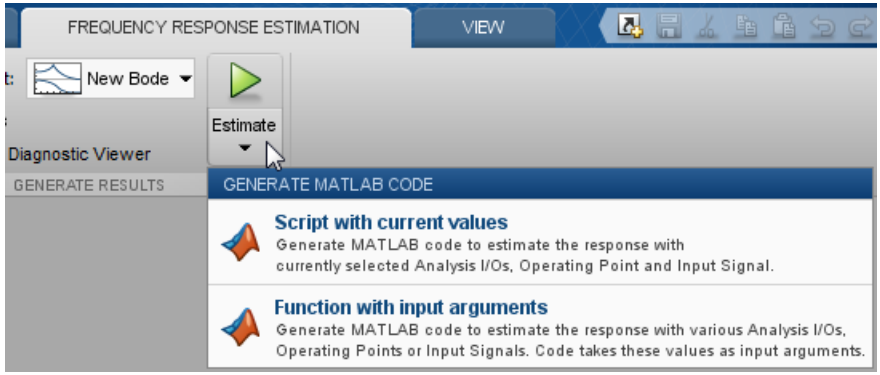
Generate MATLAB Code for Repeated or Batch Frequency Response Estimation

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, on the **Frequency Response Estimation** tab, interactively configure the input signal, analysis I/Os, operating point, and other parameters for frequency response estimation.

2. Click **Estimate** to open the **Generate Matlab Code** menu.



3. Select the type of code you want to generate:

   - `Script with current values` — 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 with input arguments` — 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" on page 3-81

"Speeding Up Estimation Using Parallel Computing" on page 3-83

"Managing Memory During Frequency Response Estimation" on page 3-87

Ways to Speed up Frequency Response Estimation

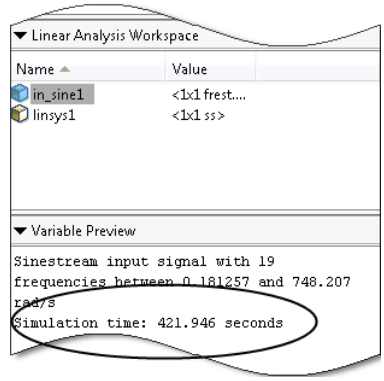
The most time consuming operation during frequency response estimation is the simulation of your Simulink model. You can try to speed up the estimation using any of the following ways:

- "Reducing Simulation Stop Time" on page 3-81
- "Specifying Accelerator Mode" on page 3-83
- "Using Parallel Computing" on page 3-83

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, select the input signal in the Linear Analysis Workspace are of the Linear Analysis Tool. 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 “Modifying Input Signals for Estimation” on page 3-24.

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?” in the Simulink documentation.

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 3-83.

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

To manually open a parallel pool that uses a specific cluster profile, use:

```
parpool(MyProfile);
```

`MyProfile` is the name of a cluster profile.

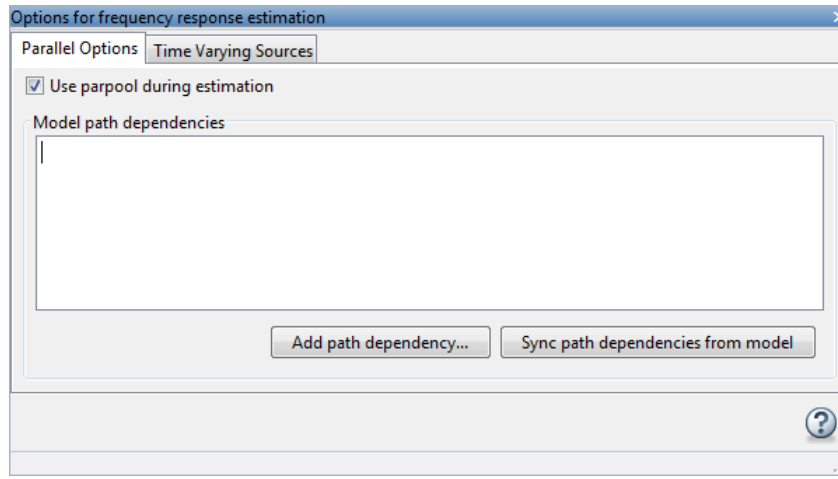
Estimating Frequency Response Using Parallel Computing Using Linear Analysis Tool

After you configure your parallel computing settings, as described in "Configuring MATLAB for Parallel Computing" on page 3-83, you can estimate the frequency response of a Simulink model using the Linear Analysis Tool.

1. In the **Frequency Response Estimation** tab of the Linear Analysis Tool, click **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 dependecy**.

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 3-83, 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 strings 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?" and "Scope of Dependency Analysis" in the Simulink documentation.

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".

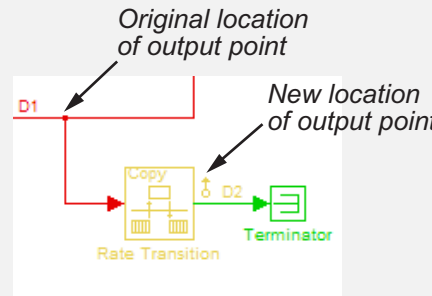
2 Try one or more of the actions listed in the following sections:

  - "Model-Specific Ways to Avoid Memory Issues" on page 3-88
  - "Input-Signal-Specific Ways to Avoid Memory Issues" on page 3-89

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". If your estimation is slow, see "Ways to Speed up Frequency Response Estimation" on page 3-81.
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 "Frequency-Response Model".

Model Type	Action
	• Use parallel computing to run the independent simulations in parallel on different computers. See "Speeding Up Estimation Using Parallel Computing" on page 3-83.

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 3-83. • Divide the input signal into multiple signals using `fselect`. Estimate the frequency response

Input Signal Type	Action
	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 3-60.

4

Designing Compensators

- "Choosing a Compensator Design Approach" on page 4-2
- "Introduction to Automatic PID Tuning" on page 4-3
- "What Plant Does the PID Tuner See?" on page 4-5
- "PID Tuning Algorithm" on page 4-6
- "Design Controllers with the PID Tuner" on page 4-7
- "Tune PID Controller to Balance Tracking and Disturbance Rejection Performance" on page 4-18
- "Design Two-Degree-of-Freedom PID Controllers" on page 4-28
- "Tune PID Controller Within Model Reference" on page 4-30
- "Troubleshooting Automatic PID Tuning" on page 4-33
- "Designing a Simulink PID Controller (2DOF) Block for a Reactor" on page 4-39
- "Designing PID Controller in Simulink with Estimated Frequency Response" on page 4-52
- "Designing a Family of PID Controllers for Multiple Operating Points" on page 4-60
- "Design and Analysis of Control Systems" on page 4-68

Choosing a Compensator Design Approach

Simulink Control Design provides two tools to tune Simulink blocks, such as Transfer function and PID Controller blocks:

- PID Tuner
- SISO Design Tool

Use the following table to determine which tool supports what you want to do.

	PID Tuner	SISO Design Tool
Supported Blocks	PID Controller PID Controller 2DOF	Linear blocks
Loop Structure	Single-loop control systems with unit feedback	Single- or multi-loop control systems
Control Design Approach	Simple automatic PID gain tuning by specifying system response time and transient response	Graphically tune poles and zeros on design plots, such as Bode, root locus, and Nichols Use a PID, LQG, IMC, Robust Control Loop Shaping, and Simulink Design Optimization automated tuning method
Analysis of Control System Performance	Time and frequency response for reference tracking and disturbance rejection	Any combination of responses for any input reference or disturbance in your Simulink model using SISO Tool LTIViewer

If you have a nonlinear plant, the software automatically linearizes your model before tuning the controller blocks.

Introduction to Automatic PID Tuning

You can use the Simulink Control Design PID Tuner to tune PID gains automatically in a Simulink model containing a PID Controller or PID Controller (2DOF) block. The PID Tuner allows you to achieve a good balance between performance and robustness for either one- or two-degree-of-freedom PID controllers.

The PID Tuner:

- Automatically computes a linear model of the plant in your model. The 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 the PID Tuner See?" on page 4-5.
- Automatically computes an initial PID design with a balance between performance and robustness. The PID Tuner bases the initial design upon the open-loop frequency response of the linearized plant. See "PID Tuning Algorithm" on page 4-6.
- Provides the PID Tuner GUI to help you interactively refine the performance of the PID controller to meet your design requirements. See "Design Controllers with the PID Tuner" on page 4-7.

You can use the 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 trade-off 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 Balance Tracking and Disturbance Rejection Performance" on page 4-18

- "Designing a Simulink PID Controller (2DOF) Block for a Reactor" on page 4-39

What Plant Does the PID Tuner See?

The 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, the 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, the 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 the PID Tuner. See “Opening the Tuner” on page 4-8.

For more information about linearization, see “Linearizing 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. The algorithm designs an initial controller by choosing a bandwidth to achieve that balance, based upon the open-loop frequency response of your linearized model. When you interactively change the response time, bandwidth, or phase margin using the PID Tuner interface, the algorithm computes new PID gains. (See “Design Controllers with the PID Tuner” on page 4-7 for information about using the PID Tuner interactively.)

Design Controllers with the PID Tuner

In this section...
"Prerequisites for PID Tuning" on page 4-7
"Opening the Tuner" on page 4-8
"Analyzing the Design in the PID Tuner" on page 4-10
"Refining the Design" on page 4-12
"Verifying the PID Design in Your Simulink Model" on page 4-13
"Tuning at a Different Operating Point" on page 4-14

Prerequisites for PID Tuning

Before you can use the 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 the PID Tuner. See "Design and Analysis of Control Systems" on page 4-68 and Cascaded Multi-Loop/Multi-Compensator Feedback Design for more information.
  - The PID Controller blocks support vector signals. However, using the 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 the Tuner

To open the 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, click **Tune** to launch the PID Tuner.

When you open the PID Tuner, the following actions occur:

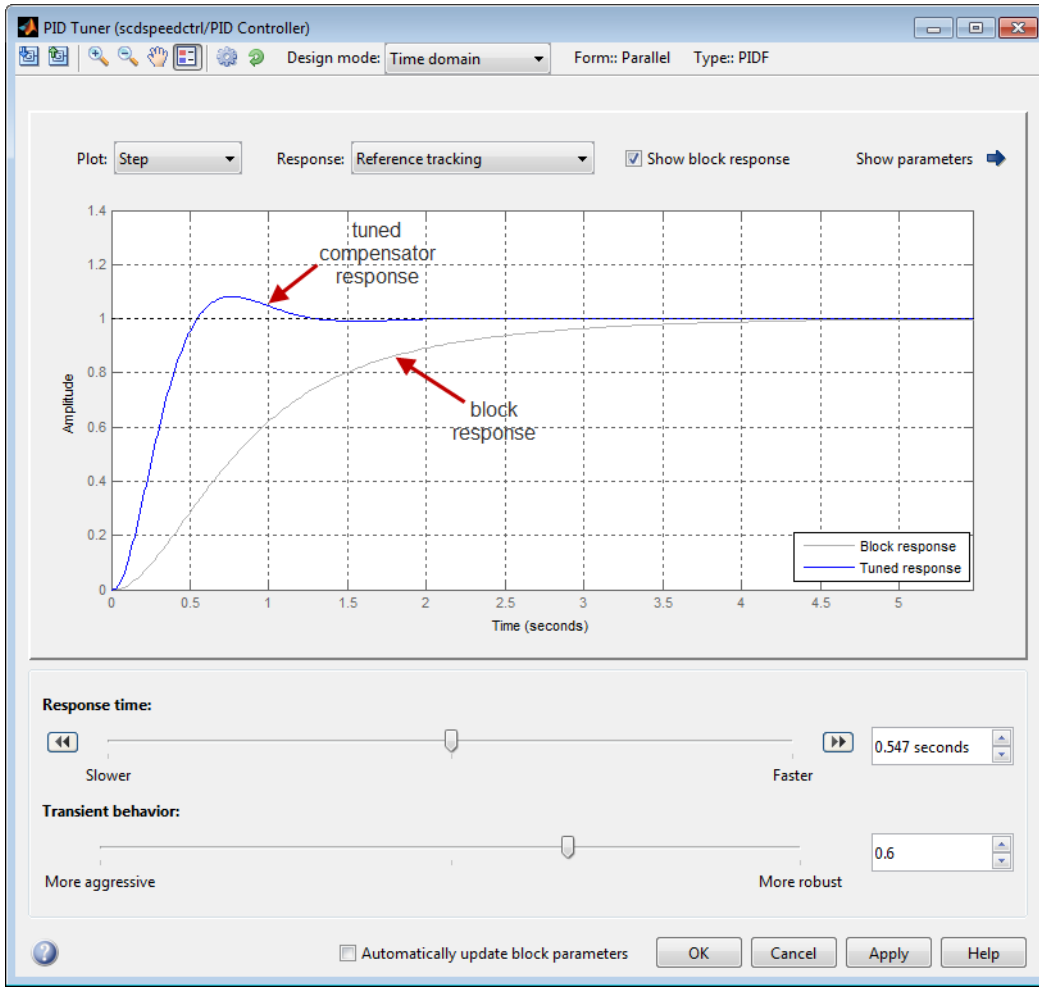
- The PID Tuner automatically linearizes the plant at the operating point specified by the model initial conditions, as described in "What Plant Does the PID Tuner See?" on page 4-5. If you want to design a controller for a different operating point, see "Tuning at a Different Operating Point" on page 4-14.

  Note If the plant model in the PID loop linearizes to zero, the 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 "Tuning at a Different Operating Point" on page 4-14).
  - 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 "Designing PID Controller in Simulink with Estimated Frequency Response" on page 4-52.

- The PID Tuner computes an initial compensator design for the linearized plant model using the algorithm described in "PID Tuning Algorithm" on page 4-6.
- The PID Tuner displays the closed-loop step reference tracking response for the initial compensator design in the PID Tuner dialog box. 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.

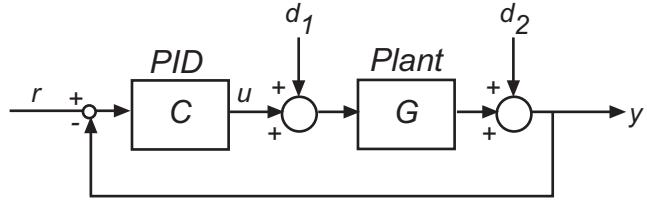


Tip After the tuner launches, you can close the PID Controller block dialog box.

Analyzing the Design in the PID Tuner

To determine whether the compensator design meets your requirements, you can analyze the system response using the response plots. Select a response plot from the **Response** drop-down menu.

The PID Tuner computes the responses based upon the following single-loop control architecture:



The following table summarizes the available responses.

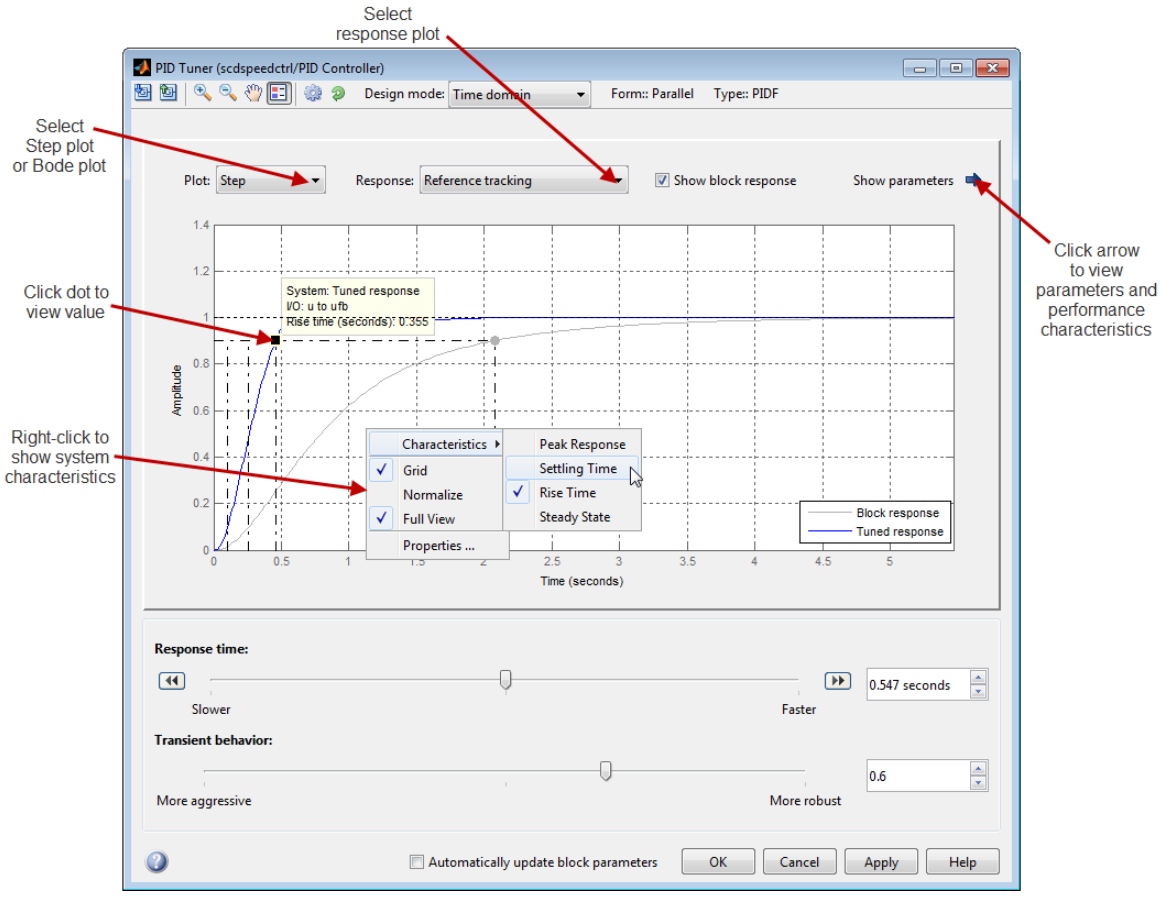
Response	Plotted System	Description
`Reference tracking`	$\frac{CG}{1+CG}$ (from r to y)	Shows the closed-loop system response to a step change in setpoint. Use when your design specifications include setpoint tracking.
`Controller effort`	$\frac{C}{1+CG}$ (from r to u)	Shows the closed-loop controller output response to a step change in setpoint. Use when your design is limited by practical constraints, such as controller saturation.
`Input disturbance Rejection`	$\frac{G}{1+CG}$ (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.

Response	Plotted System	Description
`Output disturbance Rejection`	$\frac{1}{1+CG}$ (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.
`Open-loop`	CG	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.
`Plant`	G	Shows the plant response. Use to examine plant dynamics.

Use the **Plot** drop-down menu to choose `Step` plot (time-domain response) or `Bode` plot (frequency-domain response).

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 the **Show Parameters** arrow .



Tip To perform further analysis on the plant model, click the Export plant model to workspace button to export the plant to the MATLAB workspace as an LTI object.

Refining the Design

If the response of the initial controller design does not meet your requirements, you can interactively adjust the design. The PID Tuner gives you two **Design mode** 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 domain` — 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 trade-off between reference tracking and disturbance rejection performance. For an example that shows how to use the sliders to adjust this trade-off, see "Tune PID Controller to Balance Tracking and Disturbance Rejection Performance" on page 4-18.

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 "Verifying the PID Design in Your Simulink Model" on page 4-13.

Tip To revert to the initial controller design after moving the sliders, click .

Verifying the PID Design in Your Simulink Model

In the PID Tuner, you tune the compensator using a linear model of your plant. First, you find a good compensator design in the 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, click **Apply** 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 the PID Tuner, select **Automatically update block parameters**.

2 Simulate the Simulink model, and evaluate whether the simulation output meets your design requirements.

Because the PID Tuner works with a linear model of your plant, the simulated response sometimes does not match the response in the PID Tuner. See "Simulated Response Does Not Match the PID Tuner Response" on page 4-35 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 4-36.

Tuning at a Different Operating Point

By default, the PID Tuner linearizes your plant and designs a controller at the operating point specified by the initial conditions in your Simulink model. In some cases, this operating point can differ from the operating point you want to design a controller for. 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 the PID Tuner uses for linearizing your plant and designing a controller.

To set a new operating point for the PID Tuner, use one of the following methods. The method you choose depends upon the information you have about your desired operating point:

Known State Values Yield the Desired Operating Conditions

1 Close the 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 launch the PID Tuner. The PID Tuner linearizes the plant using the new default operating point and designs a new initial controller for the new linear plant model.

After the PID Tuner generates a new initial controller design, continue from "Analyzing the Design in the PID Tuner" on page 4-10.

Your Model Has Desired Operating Conditions at a Known Time

1 Click the Design with new plant model button in the PID Tuner to open the **Obtain plant model** dialog box.

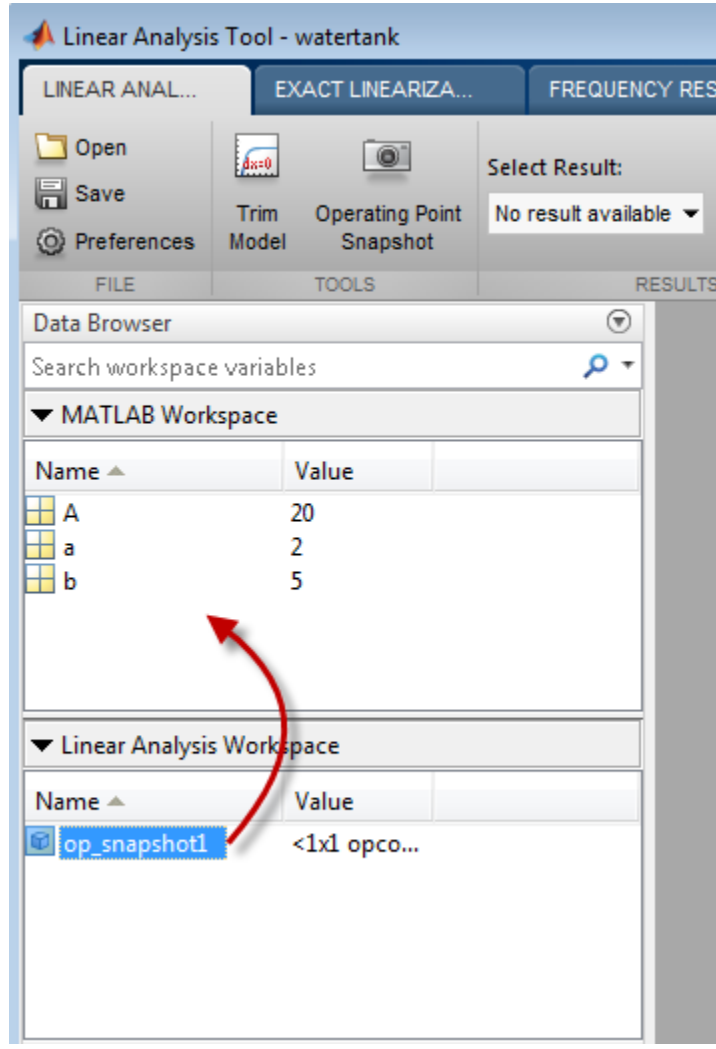
2 Select **Linearizing the Simulink model at simulation snapshot time** and enter a time value. Enter a time at which you expect the model to have the desired equilibrium operating conditions.

3 Click **OK**. The PID tuner linearizes the plant using the new operating point and designs a new initial controller for the new linear plant model.

After the PID Tuner generates a new initial controller design, continue from "Analyzing the Design in the PID Tuner" on page 4-10.

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 the PID Tuner, click to open the **Obtain plant model** dialog box.

3 Select **Importing an LTI system or linearizing at an operating point defined in MATLAB workspace**.

4 Select your exported operating point in the table. The operating point you select appears highlighted.

5 Click **OK**. The PID tuner linearizes the plant using the operating point at the snapshot time and designs a new initial controller for the new linear plant model.

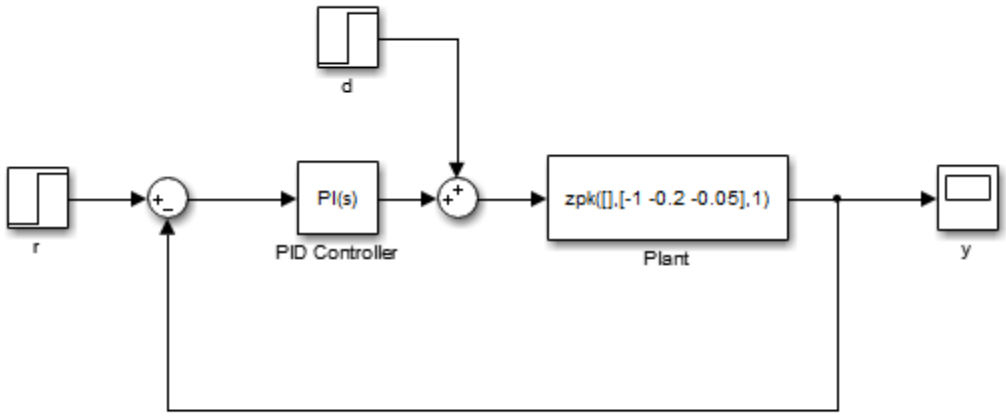
After the PID Tuner generates a new initial controller design, continue from "Analyzing the Design in the PID Tuner" on page 4-10.

Tune PID Controller to Balance Tracking and Disturbance Rejection Performance

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, the example illustrates the trade-off between reference tracking and disturbance rejection performance.

1 Load a Simulink model that contains a PID Controller block.

```
open_system('singlePIloop')
```

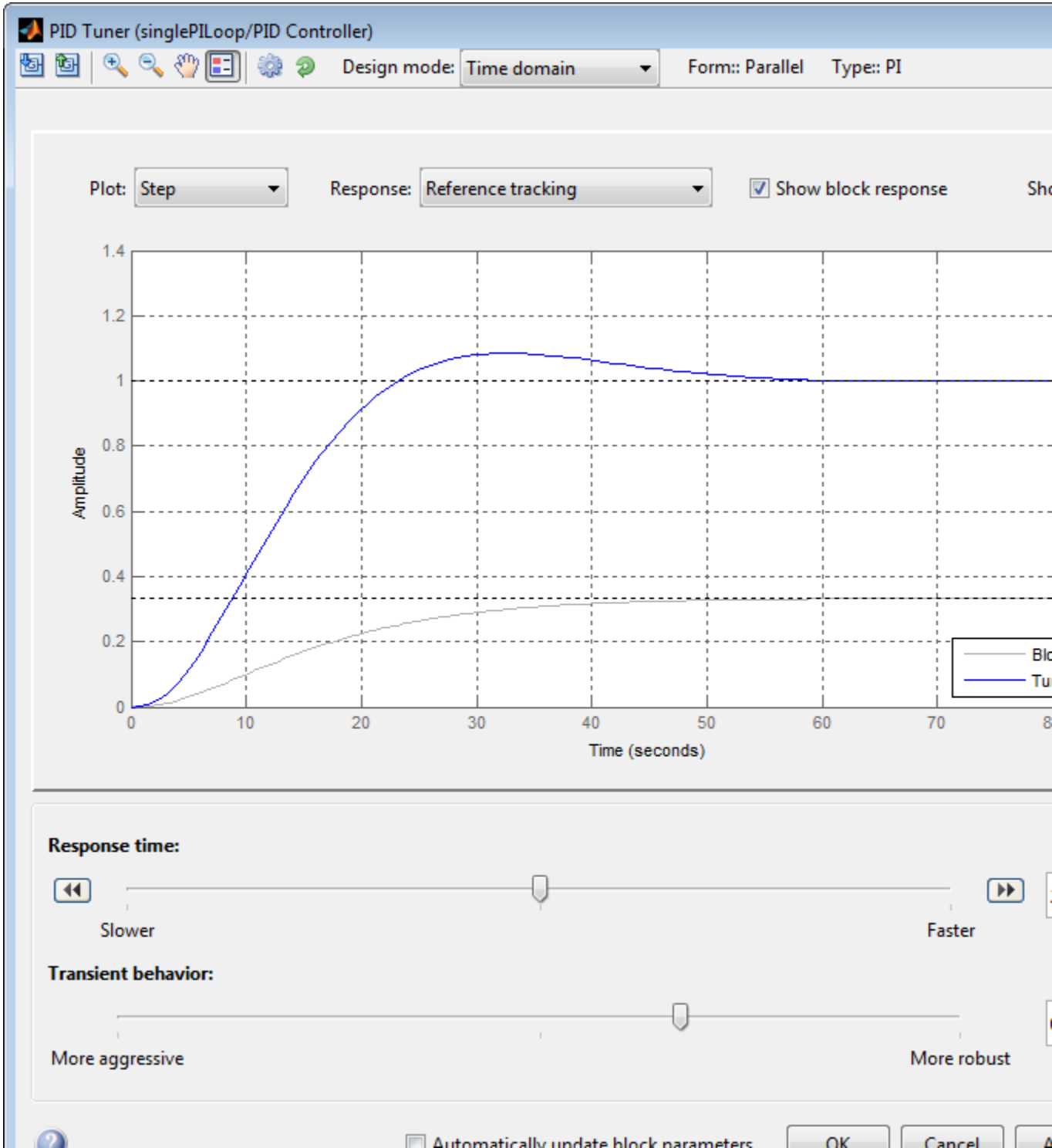


The plant has poles at $s = -1$, $s = -0.2$, and $s = -0.05$. The model also includes a reference signal and a step disturbance a the plant input.

Setpoint 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`.

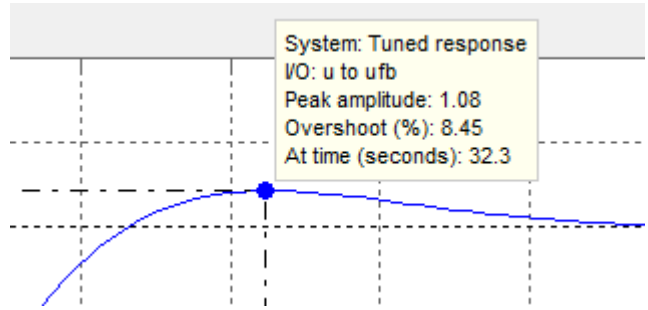
2 Double-click the PID Controller block to open the Block Parameters dialog box.

3 Click **Tune** to open the PID Tuner, which designs an initial controller for the plant.


PID Tuner (singlePILoop/PID Controller)
Design mode: Time domain
Form:: Parallel
Type:: PI
Plot: Step
Response: Reference tracking
Show block response
Amplitude
Time (seconds)
Response time:
Slower
Faster
Transient behavior:
More aggressive
More robust
Automatically update block parameters
OK
Cancel

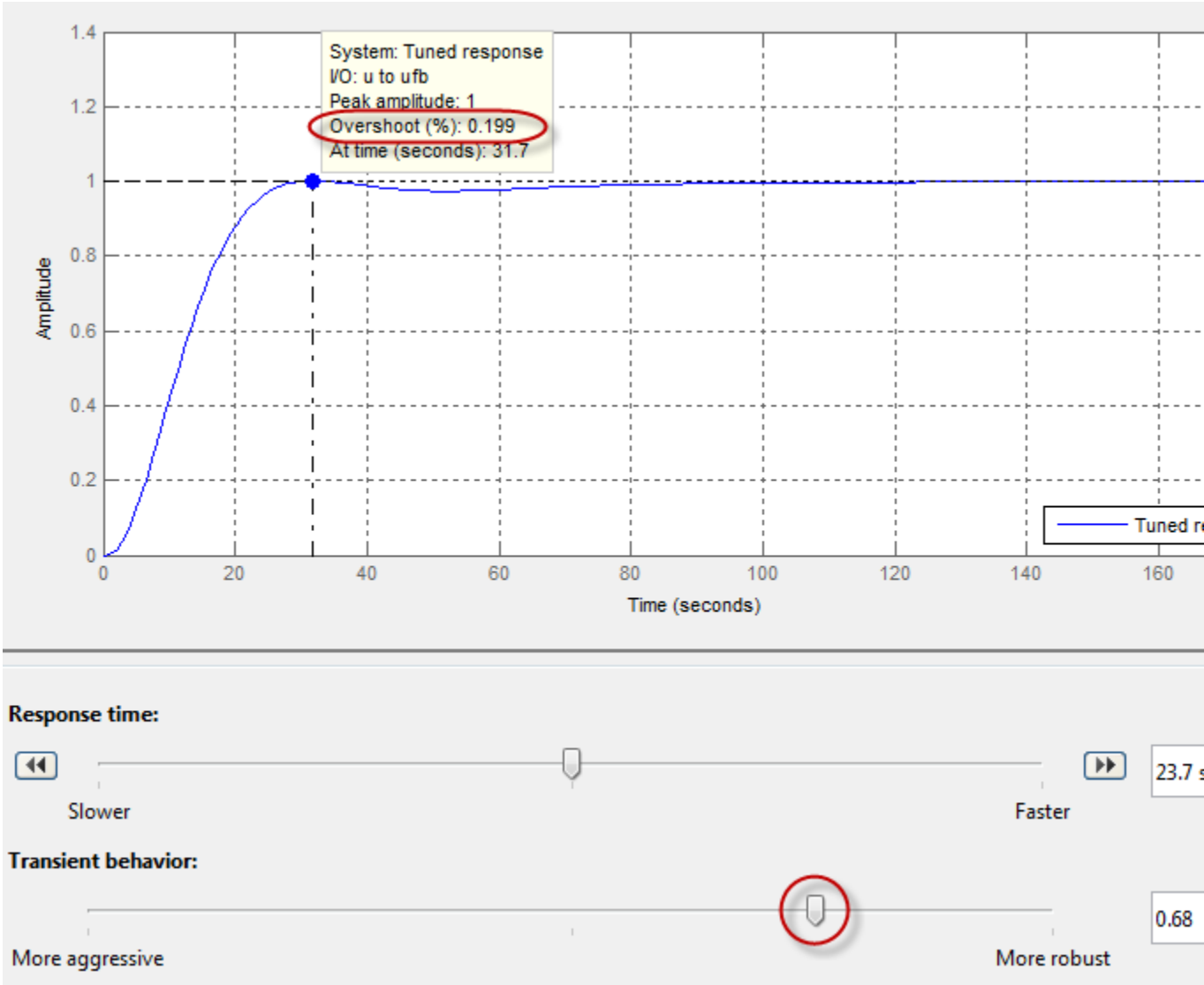
The initial controller design has some overshoot in the reference tracking response.

4. Right-click in the plot area, and select **Characteristics > Peak Response**. A blue dot appears at the point of maximum overshoot. Click the blue dot to display peak response data in a tool tip.



The tool tip shows that the overshoot is 8.45%. If this overshoot is too much for your application, you can reduce it using the **Transient behavior** slider.

5. Move the **Transient behavior** slider to the right until the overshoot is less than 0.2%.

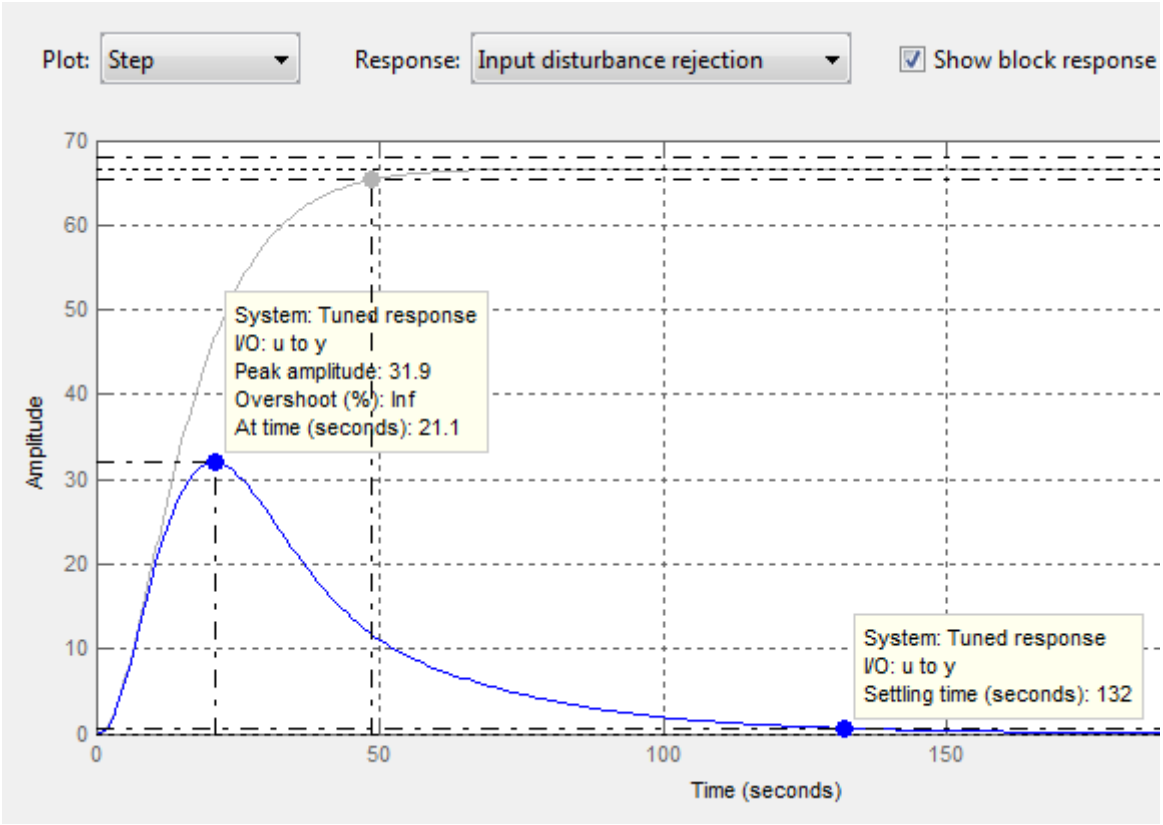


Increasing controller robustness also reduces overshoot, so moving the **Transient behavior** slider to the right improves this measure of controller performance. However, for a given controller bandwidth (or response time),

there is a trade-off between reducing reference tracking overshoot and optimizing disturbance rejection.

6 In the **Response** menu, select `Input disturbance rejection` to display the disturbance rejection response plot.

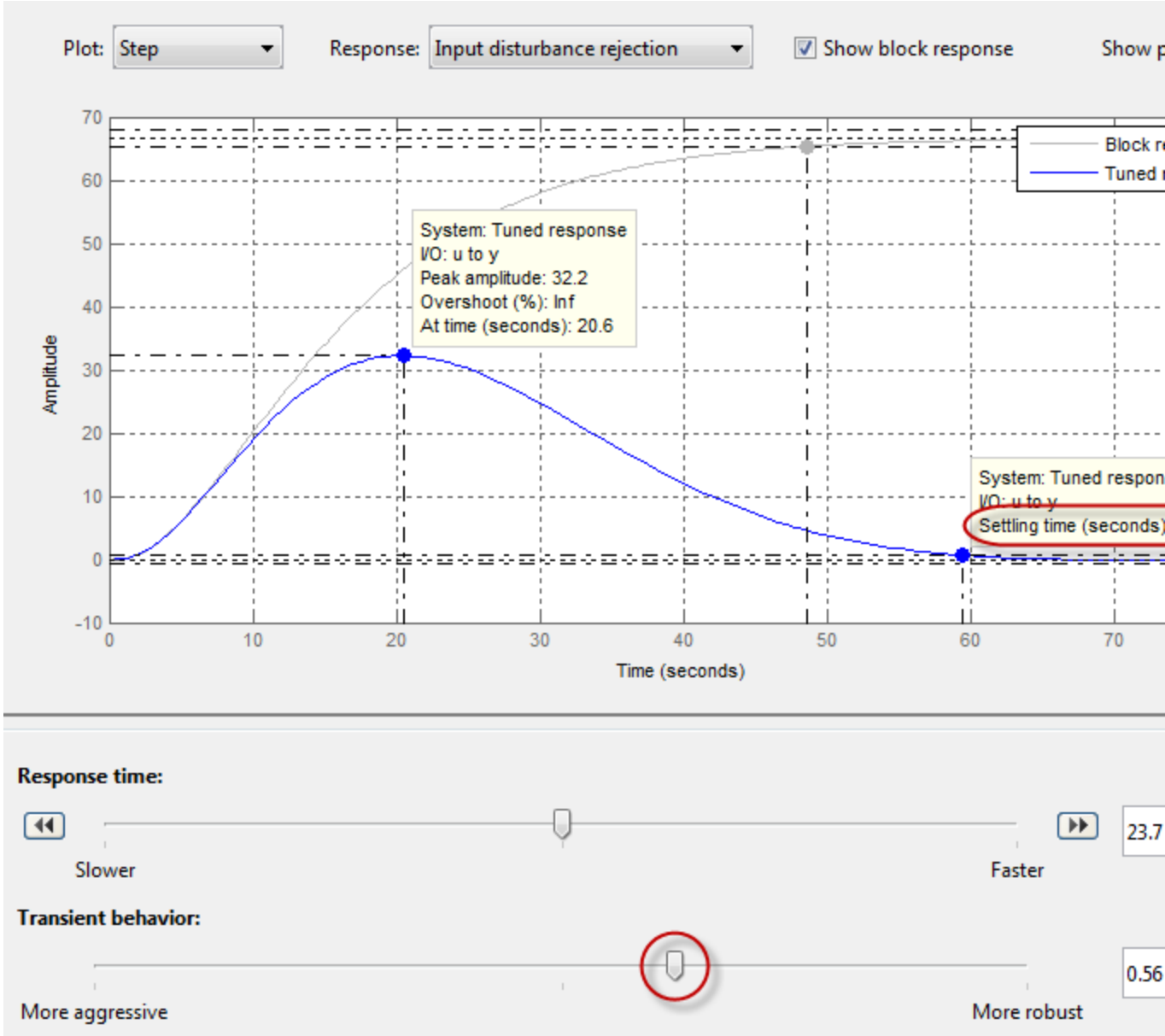
7 Right-click in the plot area, and select **Characteristics > Settling Time**. Click the blue dot to display the settling time tool tip.



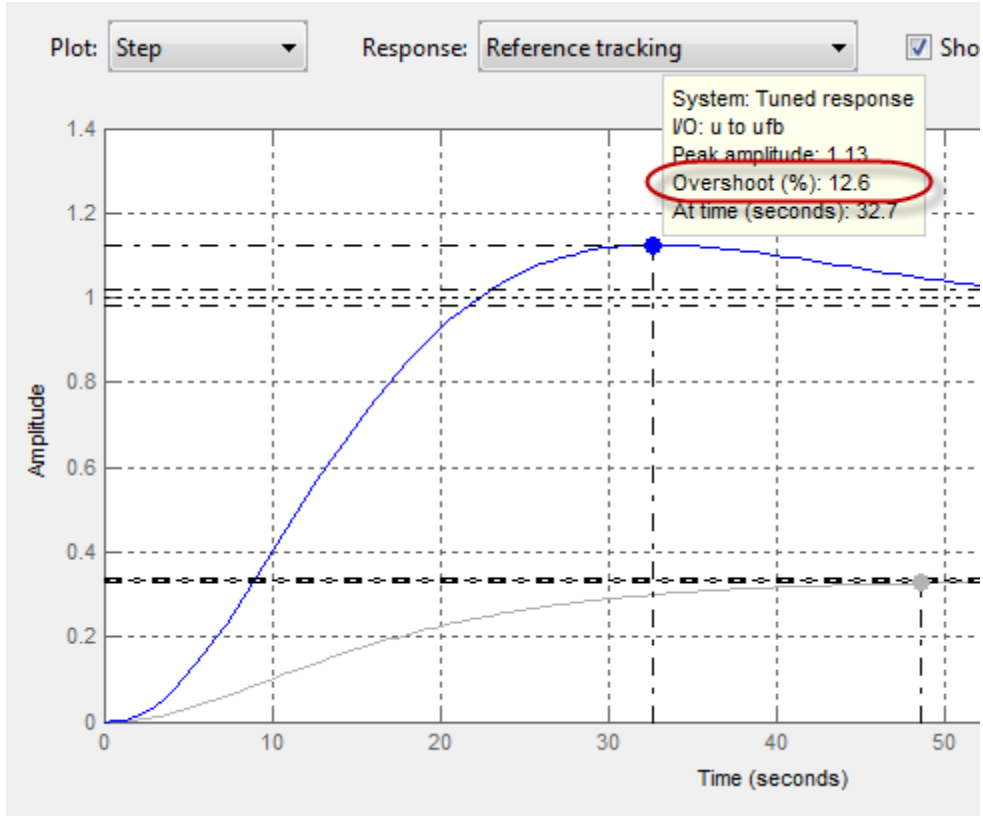
The current controller design minimizes reference tracking overshoot. However, it responds sluggishly to a disturbance at the plant input, taking over two minutes to settle. You can use the **Transient behavior** slider

to make the disturbance rejection more aggressive without changing the controller bandwidth.

8 Move the **Transient behavior** slider to the left until the disturbance response settles in under 60 seconds.


Plot: Step
Response: Input disturbance rejection
Show block response
Show p
Block r
Tuned
System: Tuned response
I/O: u to y
Peak amplitude: 32.2
Overshoot (%): Inf
At time (seconds): 20.6
System: Tuned respon
I/O: u to y
Settling time (seconds)
Amplitude
Time (seconds)
70
60
50
40
30
20
10
0
-10
0
10
20
30
40
50
60
70
Response time:
Slower
Faster
23.7
Transient behavior:
More aggressive
More robust
0.56

9 In the **Response** menu, select `Reference tracking` to see the effect of more aggressive transient behavior on the reference tracking overshoot.



Optimizing the disturbance rejection response for the given bandwidth increases the reference tracking overshoot to 12.6%.

The PID Tuner selects an initial controller design that balances this trade-off. The initial controller design has less reference tracking overshoot than the design optimized for disturbance rejection. Likewise, it has somewhat faster disturbance rejection than the design optimized to reduce overshoot. You can use the **Transient behavior** slider to adjust this balance as needed to suit your application.

To obtain independent control over reference tracking and disturbance rejection, you can use a two-degree-of-freedom controller (PID Controller (2DOF) block) instead of a single degree-of-freedom controller.

Related Examples

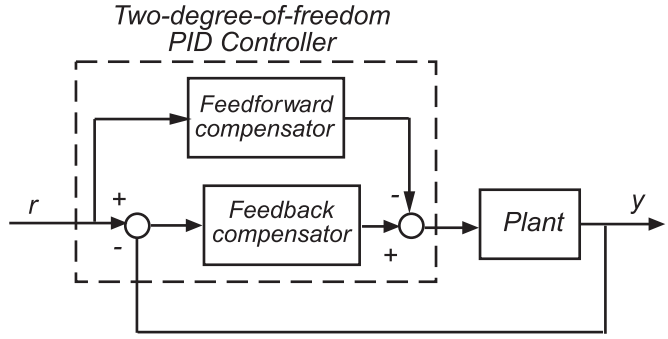
- "Design Two-Degree-of-Freedom PID Controllers" on page 4-28

Design Two-Degree-of-Freedom PID Controllers

Use the PID Tuner to tune two-degree-of-freedom PID Controller (2DOF) blocks to achieve both good setpoint tracking and good disturbance rejection.

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

The PID Tuner tunes the PID gains `P`, `I`, `D`, and `N`. The tuner does not automatically tune the setpoint weights `b` and `c`. However, you can use the PID Tuner to tune a two-degree-of-freedom PID controller by the following process:

1. Use the PID Tuner to tune the PID gains `P`, `I`, `D`, and `N` to meet your disturbance rejection requirement.

To tune this portion of the compensator, follow the procedure for tuning a one-degree-of-freedom PID compensator, as described in "Analyzing the Design in the PID Tuner" on page 4-10. and "Refining the Design" on page 4-12. Focus on the disturbance rejection plot to make sure that the tuned controller meets your disturbance rejection requirements.

2 After you have tuned the PID gains `P`, `I`, `D`, and `N`, update the PID Controller (2DOF) block with the tuned parameters. To update the block, click **Apply** in the PID Tuner, or select the **Automatically update block parameters** check box.

3 Adjust the setpoint weights `b` and `c` of the feedforward portion of the compensator to meet your setpoint tracking requirements as follows:

In the PID Controller (2DOF) block dialog box, enter values for the setpoint weights `b` and `c` between `0` and `1`.

To reduce undesirable controller response to sudden changes in the reference signal (derivative kick), set `c` to 0. Typically, give `b` a value in the range 0-1. Smaller `b` values generally result in slower reference tracking. However, `b` and `c` values do not affect loop stability or disturbance rejection.

4 Evaluate whether the compensator design meets your design requirements by viewing a simulation of the Simulink mode as described in "Verifying the PID Design in Your Simulink Model" on page 4-13.

Tune PID Controller Within Model Reference

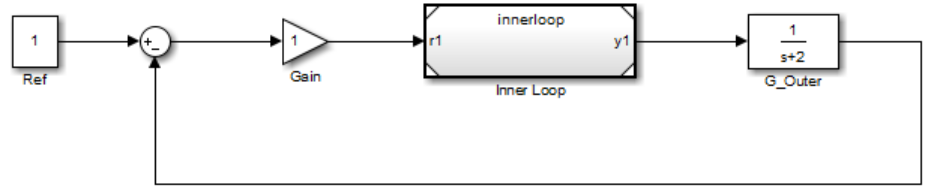
This example shows how to tune a PID controller block contained in a referenced model.

When you launch the PID Tuner from a PID controller block in a model that is referenced in one or more open models, the software prompts you to specify which open model is the top-level model for linearization and tuning. The referenced model must be in normal mode.

For more information about model referencing, see "Overview of Model Referencing" in the Simulink documentation.

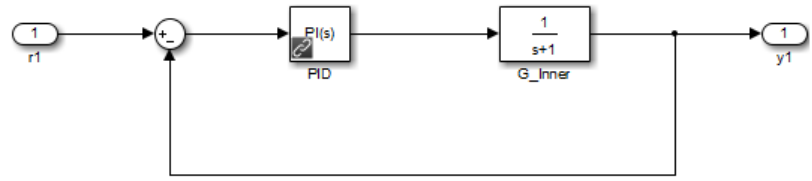
1 Open the model.

```
open('model_ref_pid');
```



The block `Inner Loop` is a referenced model that contains the PID Controller block to tune.

2 Double-click `Inner Loop` to open the referenced model.

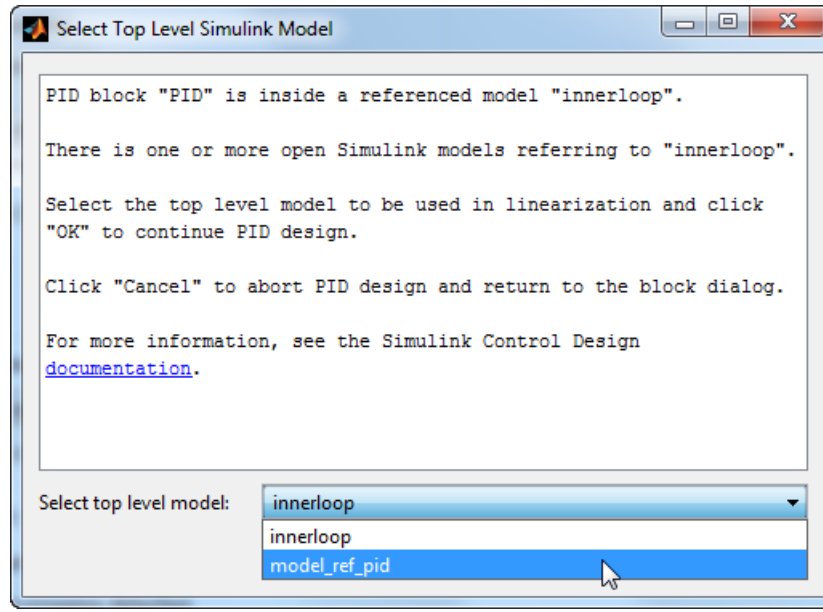


The referenced model `innerloop` contains a PID controller block, `PID`.

3 Double-click the PID controller block `PID` to open the block dialog box.

4 Click **Tune** in the block dialog box.

The software prompts you to select which open model is the top-level model for linearization and tuning.



Note The software only identifies open models containing the model reference. The PID Tuner does not detect models that contain the model reference but are not open.

Selecting `innerloop` causes the PID Tuner to disregard `model_ref_pid`. Instead, the PID 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 PID 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`.

5 Select the desired top-level model, and click **OK**.

The PID Tuner linearizes the selected model and launches. Proceed with analyzing and adjusting the tuned response as described in “Design Controllers with the PID Tuner” on page 4-7.

Troubleshooting Automatic PID Tuning

This section explains some procedures that can help you obtain better results from the PID Tuner if the basic procedures yield unsatisfactory controller performance.

In this section...

"Plant Model Linearizes to Zero" on page 4-33

"Cannot Find a Good Design in the PID Tuner" on page 4-34

"Simulated Response Does Not Match the PID Tuner Response" on page 4-35

"Cannot Find an Acceptable PID Design in the Simulated Model" on page 4-36

"Controller Performance Deteriorates When Switching Time Domains" on page 4-37

"When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain" on page 4-38

Plant Model Linearizes to Zero

If the plant model in the PID loop linearizes to zero, the 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 "Tuning at a Different Operating Point" on page 4-14).
- 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 "Designing PID Controller in Simulink with Estimated Frequency Response" on page 4-52.

Cannot Find a Good Design in the PID Tuner

What This means

You have adjusted the PID Tuner sliders, but 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 the PID Tuner to tune the parameters for the new controller type.

If you cannot find any satisfactory controller with the PID Tuner, PID control possibly is not sufficient for your requirements. You can design more complex controllers using the SISO Design Tool. For more information, see “Design and Analysis of Control Systems” on page 4-68.

Simulated Response Does Not Match the PID Tuner Response

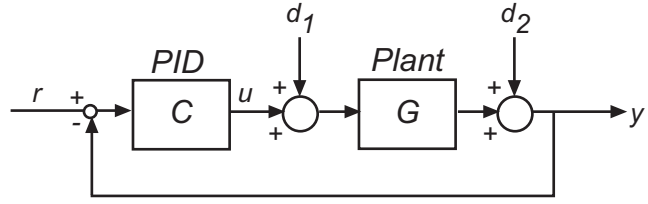
What This Means

When you run your Simulink model using the PID gains computed by the 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 4-36.

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 the 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 the PID Tuner designs for. The PID Tuner assumes the loop configuration shown in the following figure.



  As the figure illustrates, the PID Tuner designs for a PID 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. The PID Tuner

ignores nonlinear settings in the PID Controller block, which can cause the 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 the PID Tuner that is different from the operating point saved in the model. In this case, the 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 1-43.

Cannot Find an Acceptable PID Design in the Simulated Model

What This Means

You tune the PID Controller using the PID Tuner and run your Simulink model with the tuned PID gains. However, the simulated response of your model does 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 using a more complex controller. See "Design and Analysis of Control Systems" on page 4-68.

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.

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 the PID Tuner. See “Refining the Design” on page 4-12.

If you still cannot find an acceptable controller with antiwindup circuitry enabled in the PID Controller block, consider using a more complex controller. See “Design and Analysis of Control Systems” on page 4-68.

Controller Performance Deteriorates When Switching Time Domains

What This Means

You obtain a well-tuned continuous-time PID controller. Then, you convert the controller time domain using the **Time Domain** selector button in the PID Controller block dialog box. The controller performs poorly or even becomes unstable when you convert the controller to discrete time.

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, the 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, the PID Tuner resamples the plant to match the controller. All conversions use the `tustin` method (see "Continuous-Discrete Conversion Methods" in the *Control System Toolbox User's Guide*).

When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain

What This Means

When you use the PID Tuner to design a controller, the resulting derivative gain D can have a different sign from the integral gain I. The 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:

$$c = P\left(1 + \frac{1}{s} + \frac{Ds}{\frac{s}{N} + 1}\right) = P\frac{(1+DN)s^2 + (I+N)s + IN}{s(s+N)}$$

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.

Designing a Simulink PID Controller (2DOF) Block for a Reactor

This example shows how to use PID Tuner to tune a Simulink PID Controller (2DOF) block.

Introduction of the PID Controller (2DOF) Block

With a 2DOF PID controller, also known as ISA-PID controller, you can achieve good performance for both reference tracking and disturbance rejection. It contains a standard PID controller in the feedback loop and adds a pre-filter to the reference signal. The pre-filter helps produce a smoother transient response to set-point changes. In this example, you use a Simulink PID Controller (2DOF) block to control a continuous stirred tank reactor (CSTR) and you design this 2DOF PID controller in the PID Tuner.

A typical design workflow with the PID Tuner involves the following tasks:

(1) Launch the PID Tuner. When launching, the software automatically computes a linear plant model from the Simulink model and designs an initial controller.

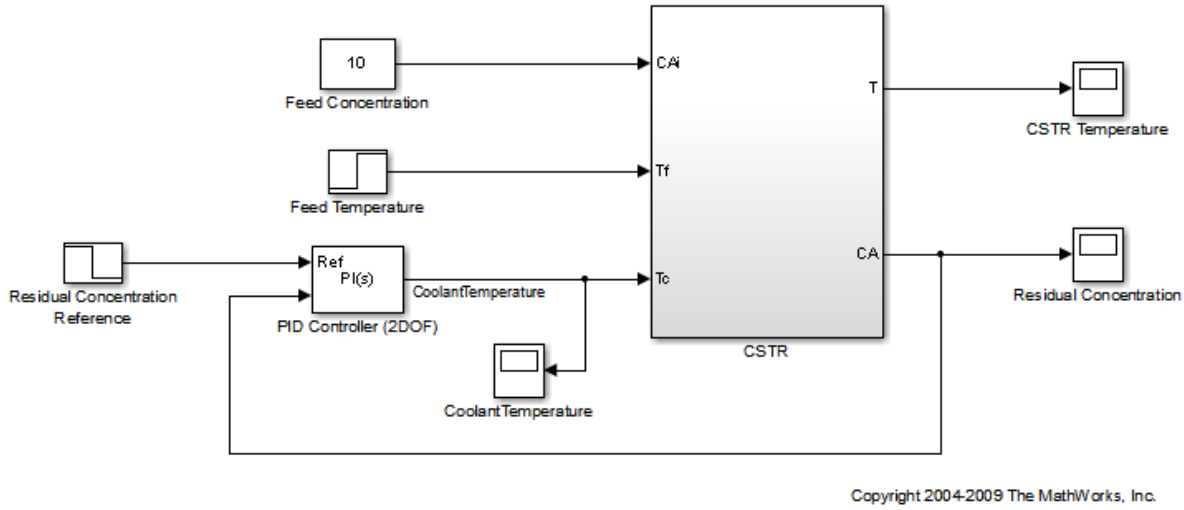
(2) Tune the controller in the PID Tuner by manually adjusting design criteria in two design modes. The tuner computes PID parameters that robustly stabilize the system.

(3) Export the parameters of the designed controller back to the PID Controller block and verify controller performance in Simulink.

Opening the Model

Open the CSTR control model and take a few moments to explore it.

```
open_system('scdcstrctrlpidblock');
```



The CSTR plant is initialized at an equilibrium operating point. The nominal value of the residual concentration is 1.96 mol/L, which is the initial condition of the `Residual Concentration Reference` block and the `Integrator1` block in the `CSTR` subsystem.

The initial condition of the integrator **I0** in the PID controller block is determined by the equilibrium operating point. In this example, since we have a PI controller in parallel form,

```
I0 = u0 - ((b-1)*y0*P)
```

where **u0** is the steady state controller output (300), and **y0** is the steady state plant output (1.96). Since **b** is 1, **I0** = **u0** = 300.

For background, see Seborg, D.E. et al., "Process Dynamics and Control", 2nd Ed., 2004, Wiley, pp.34-36.

Design Overview

In this example, control the residual concentration of the CSTR by manipulating reactor coolant temperature. The overall design requirements are:

- Track a sudden decrease of 0.5 in the reference signal from a Simulink step block `Residual Concentration Reference`. The detailed design requirements are:

  ```
  Settling time under 10 seconds

  Zero steady-state error to the step reference input

  Overshoot below 0.1
  ```

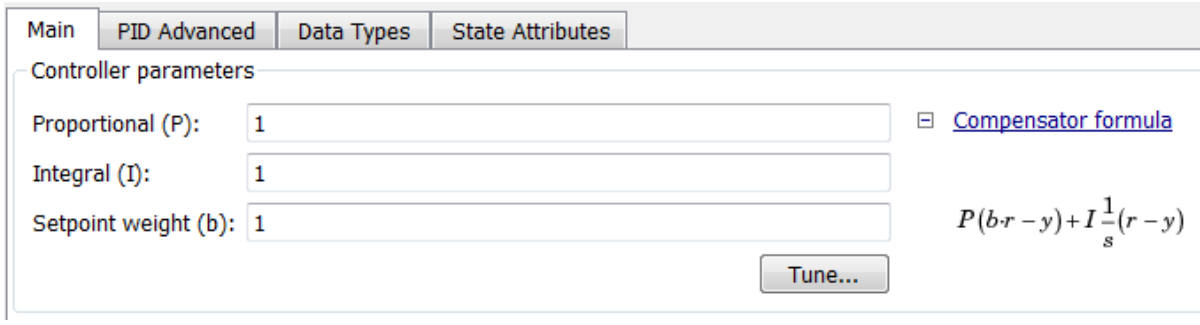
- Reject a 5 degree sudden increase in the feed temperature from a Simulink step block `Feed Temperature`. The detailed design requirements are:

  ```
  Settling time under 10 seconds

  Peak deviation from steady state below 0.01
  ```

In this example you design a PI controller in the PID Tuner to achieve good responses in both reference tracking and disturbance rejection.

Opening the PID Tuner

To launch the PID Tuner, double-click the PID Controller block to open its block dialog. In the **Main** tab, click **Tune**.



After you launch the PID Tuner, close the PID block dialog and move the PID Tuner beside the Simulink model. Also, open the residual concentration scope window.

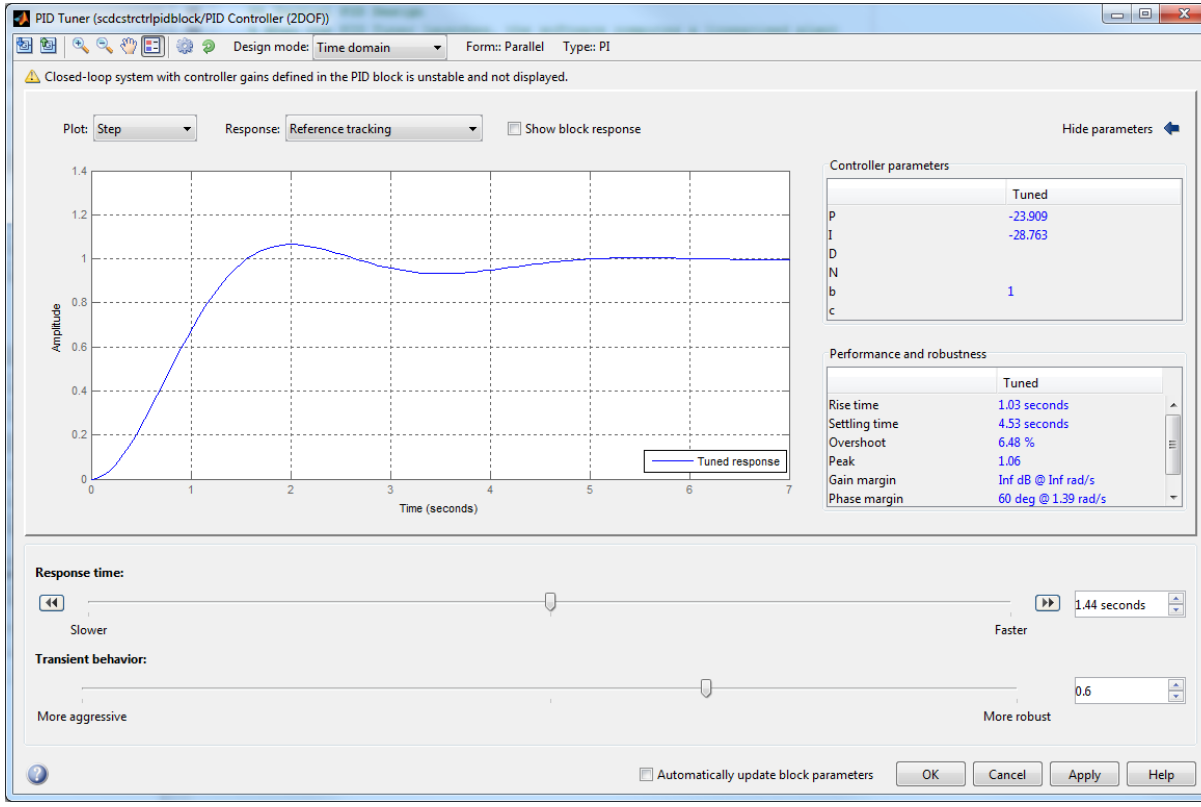
Initial PID Design

When the PID Tuner launches, the software computes a linearized plant model. The software automatically identifies the plant input and output, and uses the current operating point for the linearization. The plant can have any order and can have time delays.

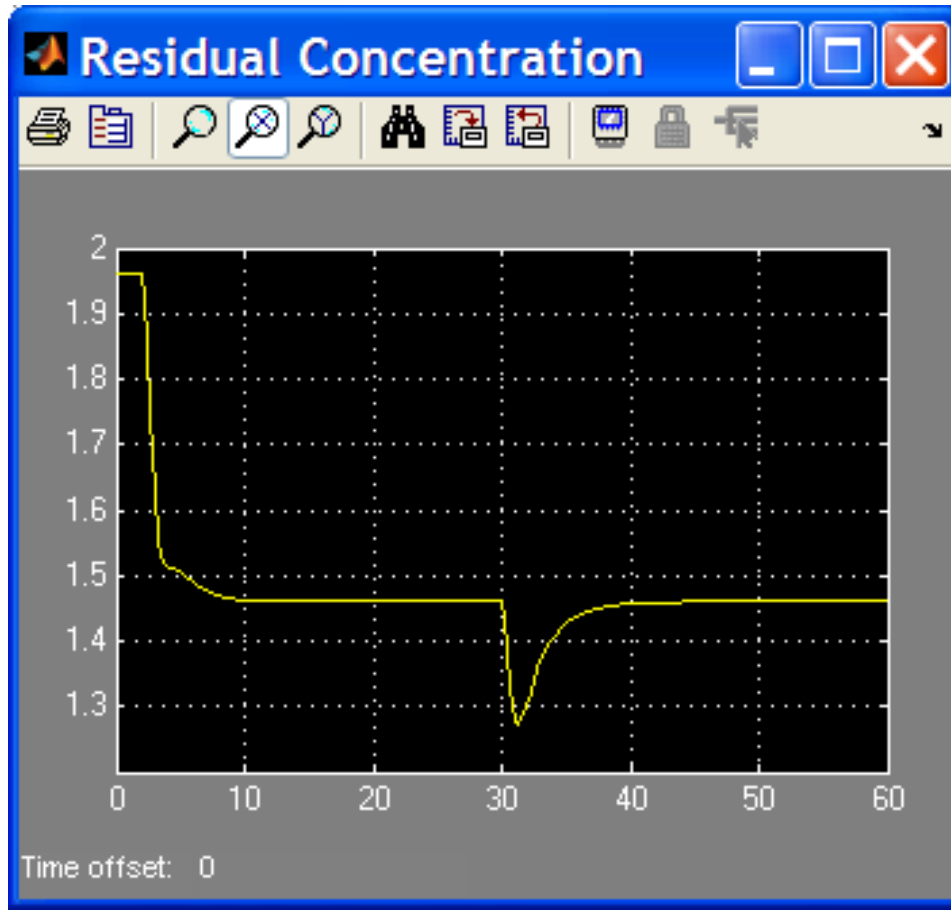
The PID Tuner computes an initial PI controller to achieve a reasonable tradeoff between performance and robustness. By default, step reference tracking performance displays in the plot.

Click the **Show parameters** arrow to view controller parameters P, I, b, and a set of performance and robustness measurements. In this example, the initial PI controller design gives a settling time of 4.4 second, which meets the requirement.

The following figure shows the PID Tuner dialog with initial design:



To test the initial design on the nonlinear model, click **Apply** in the PID Tuner. This writes the parameters back to the PID block in the Simulink model. Run a simulation and view the closed-loop response:



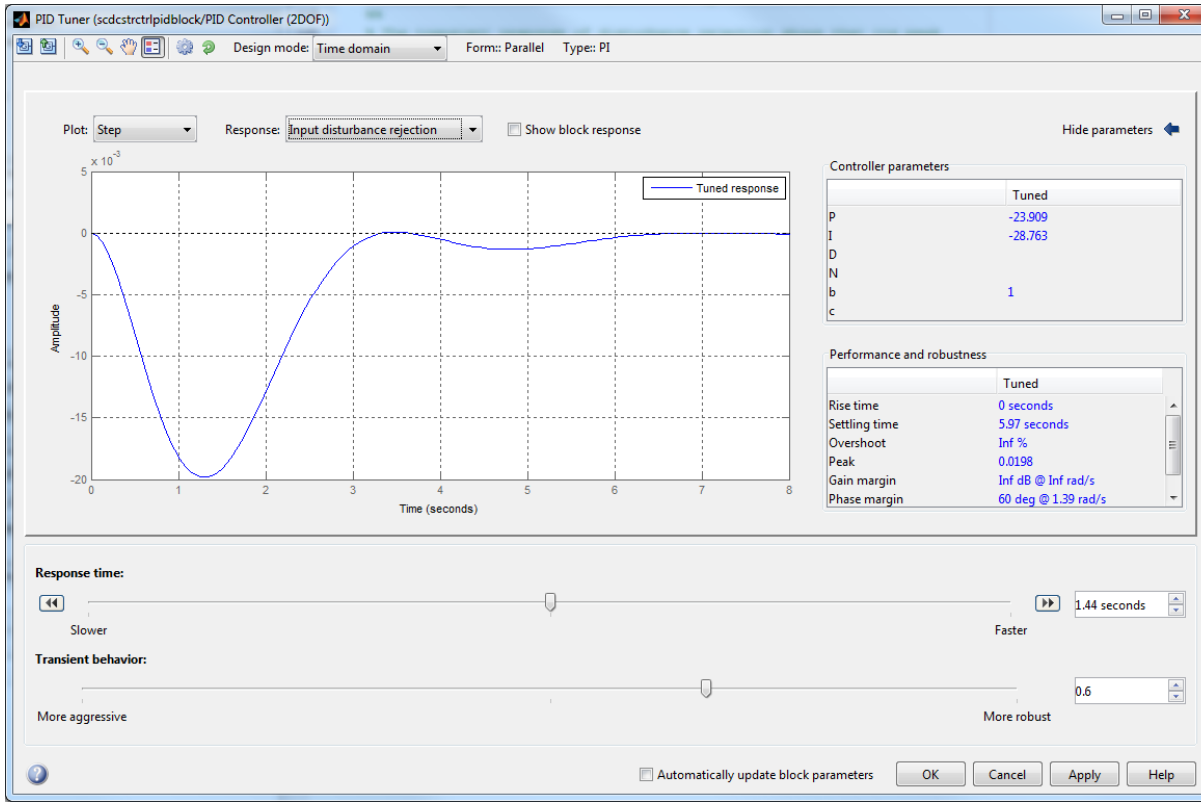
The transient response of disturbance rejection shows that its peak deviation is about 0.2, which exceeds the design requirement. You need to reduce the peak deviation by at least 50% using the PID Tuner.

Design for Disturbance Rejection in the Time Domain Design Mode

The PID Tuner provides step plot for different loop responses such as reference tracking (from r to y) and controller efforts (from r to u), etc. In this example since the disturbance occurs at the reactor feed temperature, the

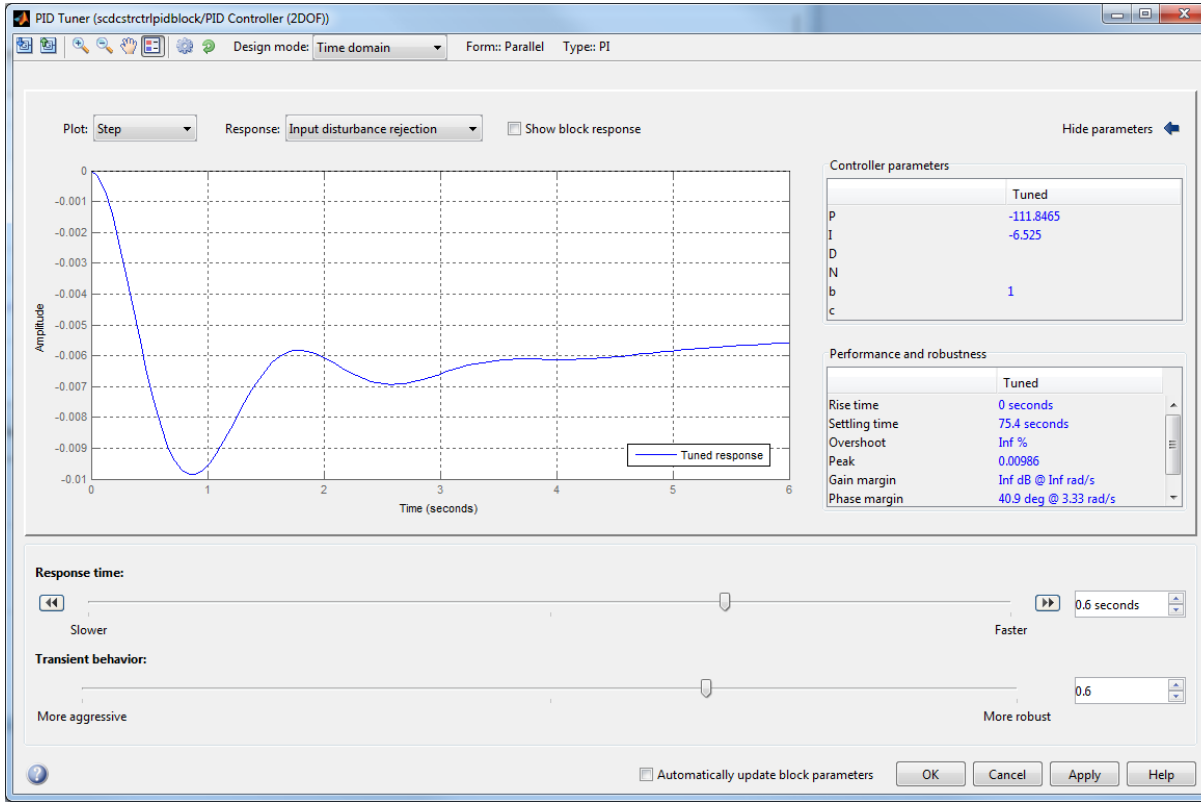
closest plot you can get is the input disturbance rejection plot that assumes a step disturbance occurs at the input of the plant model.

The following figure shows the disturbance rejection performance with initial controller:



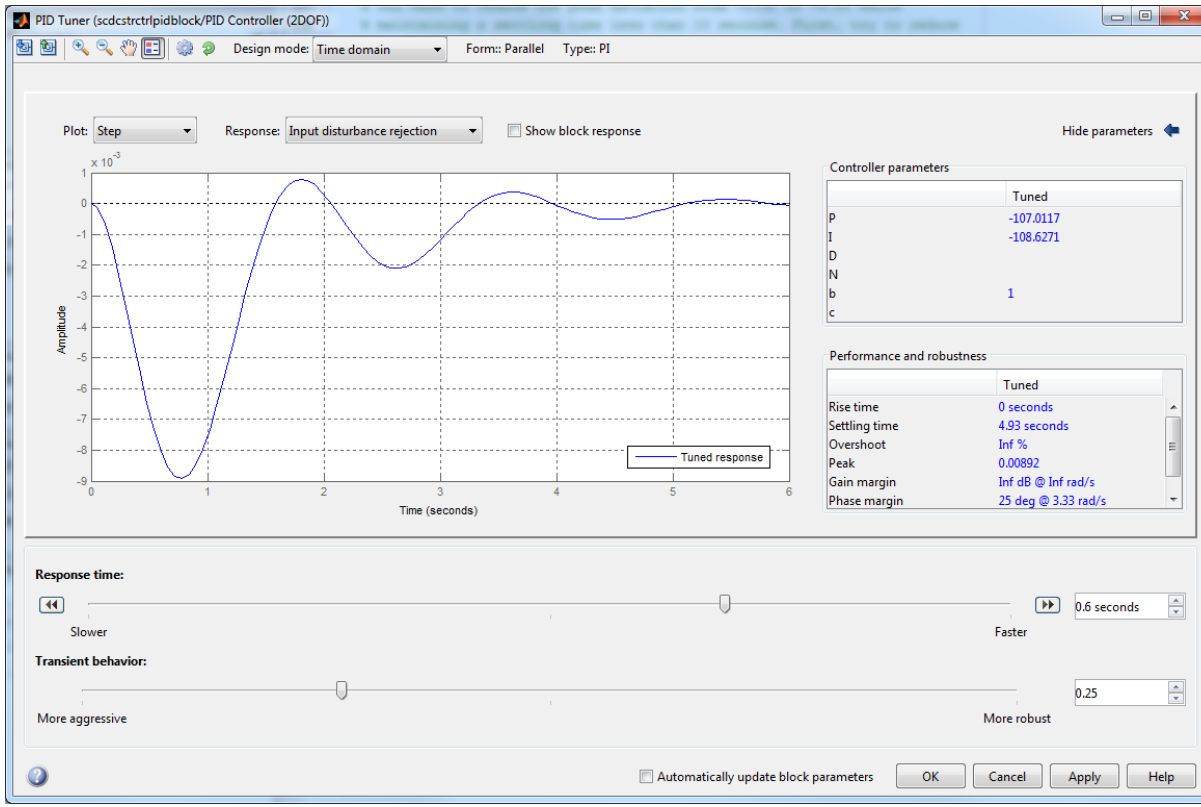
You need to reduce the peak deviation from -0.02 to -0.01 while maintaining a settling time less than 10 seconds. First, try to reduce the peak deviation by decreasing the response time, which can be achieved by moving the response time slider to the right.

The following figure shows the peak deviation is less than 0.01 when response time is reduced to 0.6 second:

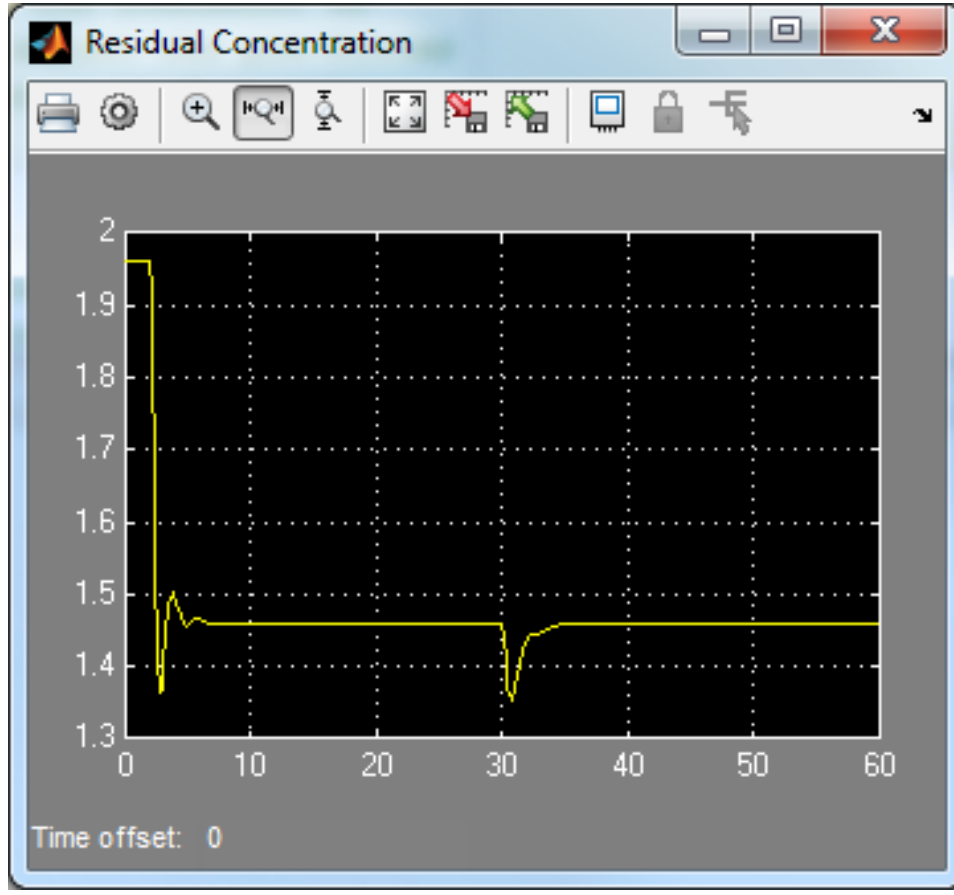


But the settling time now significantly exceeds 10 seconds. To reduce the settling time below 10 seconds, you also need to make the transient behave more aggressively, which can be achieved by moving the transient behavior slider to the left.

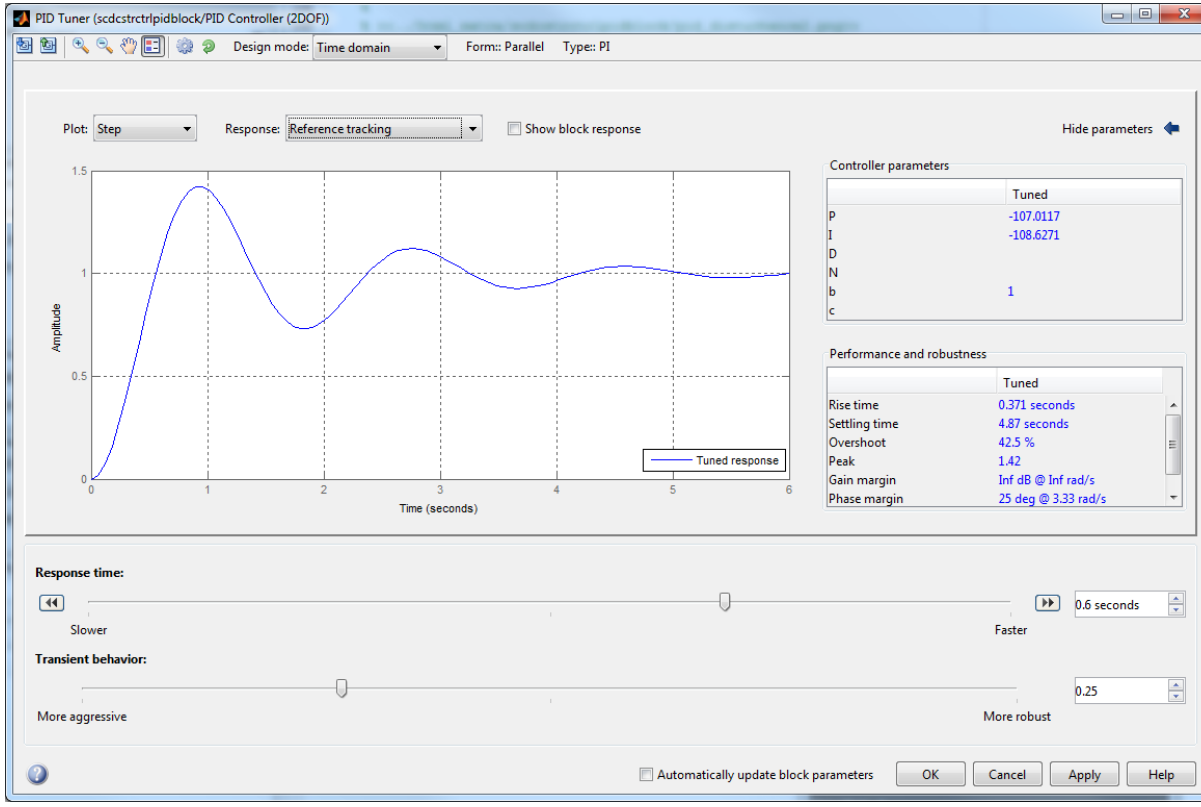
The following figure shows the final disturbance rejection performance when the response time is 0.6 seconds and the transient behavior value is 0.25.



To test the new design on the nonlinear model, click **Apply** in the PID Tuner. Run a simulation and view the closed-loop response:



The disturbance rejection performance satisfies the requirements. However, because the controller is very aggressive, the overshoot of reference tracking now exceeds the limit as shown in the figure below.



You need to adjust the pre-filter in the 2DOF PID block to improve the reference tracking performance.

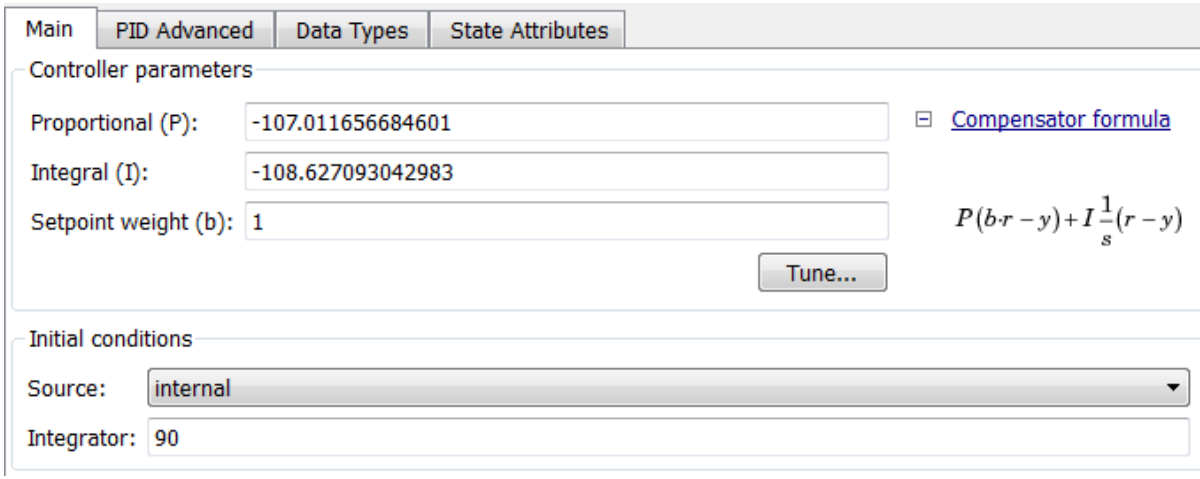
Completing PID Design with Set-point Weighting

The parameter **b** in an ISA PI or PID controller is the set-point weight on the reference signal feeding into the proportional gain of the controller. It has a range between 0 and 1, and its default value is 1. By reducing its value, the reference tracking performance becomes smoother. In this example, open the PID block dialog and set **b** to 0.

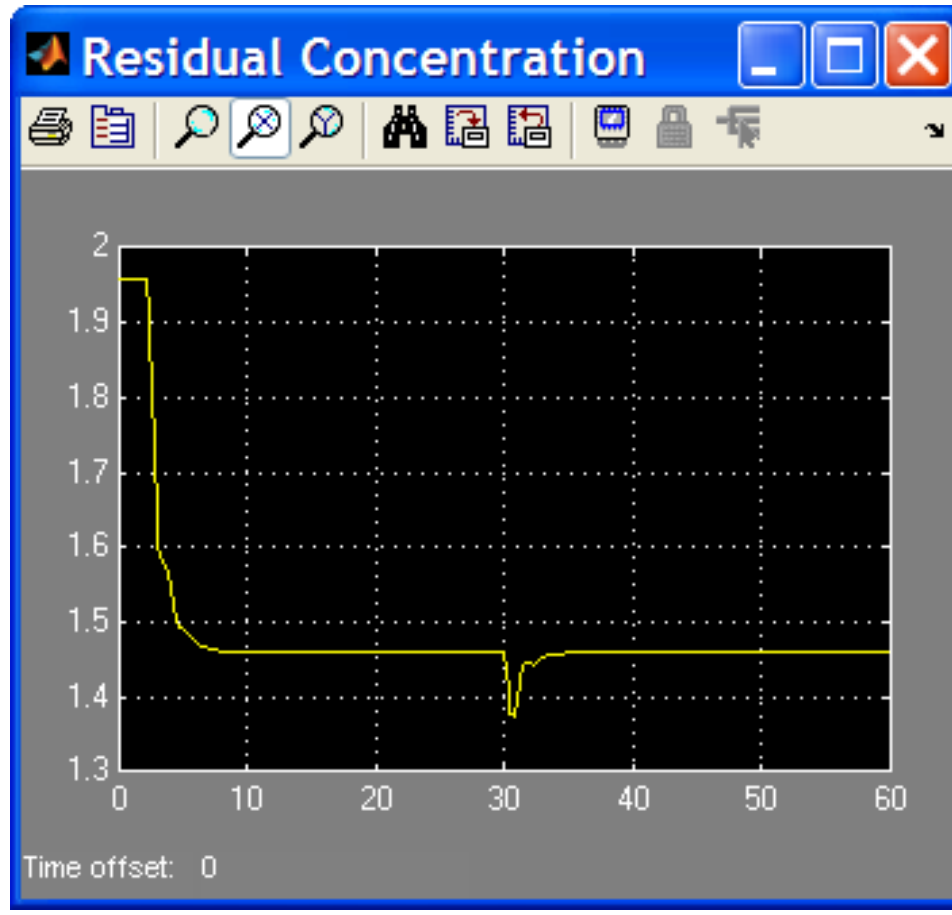
Because **b** and **P** are changed, you need to adjust the initial condition of the integrator **I0** in the PID controller block to make sure the initial operating point is still at equilibrium. The relationship is:

```
I0 = u0 - ((b-1)*y0*P)
```

where **u0** is 300 and **y0** is 1.96. With **P** from PID Tuner (-107) and **b** from the block dialog (0), **I0** becomes 90. Update the initial condition of the integrator with this value in the block dialog and the simulation will start at the equilibrium operating point.



The following figure shows that there is no overshoot in reference tracking with the updated design.



In summary, you can design a 2DOF PID controller in two steps to achieve balanced performance in reference tracking and disturbance rejection. The first step is using the PID Tuner to obtain good load disturbance rejection; and the second step is manually adjusting set-point weights **b** and **c** in the block dialog to obtain good reference tracking. Notice that changing **b** and **c** does not affect closed-loop stability or load disturbance rejection performance.

```
bdclose('scdcstrctrlpidblock')
```

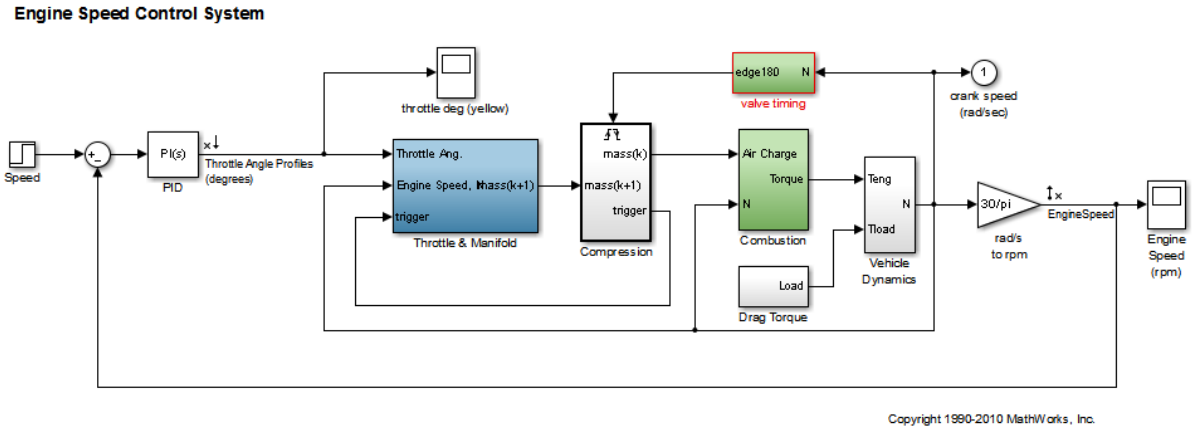
Designing PID Controller in Simulink with Estimated Frequency Response

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

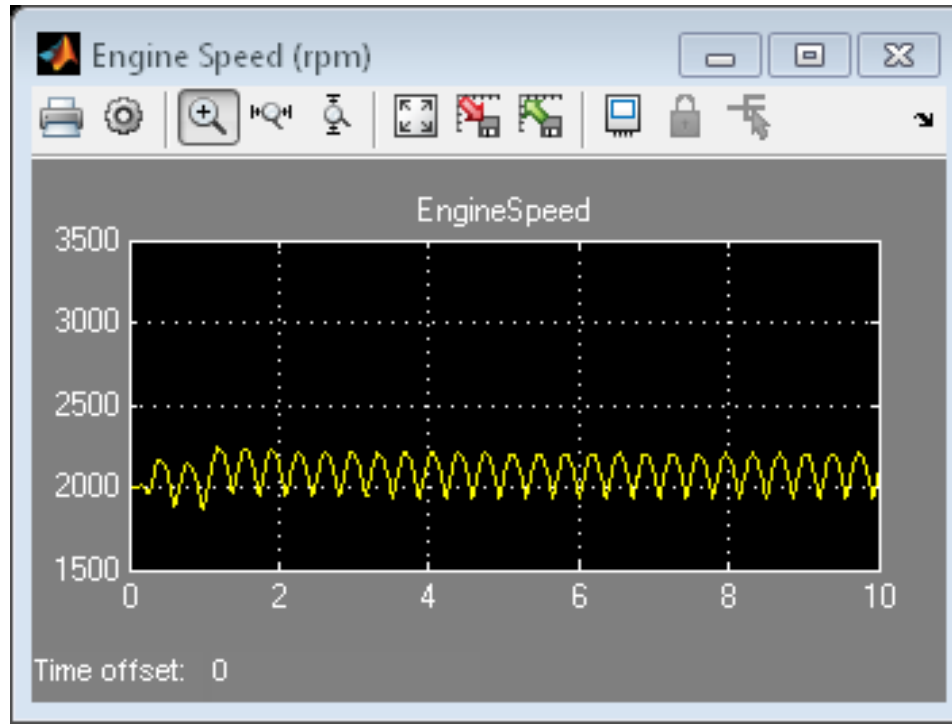
Open the engine control model and take a few moments to explore it.

```
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.

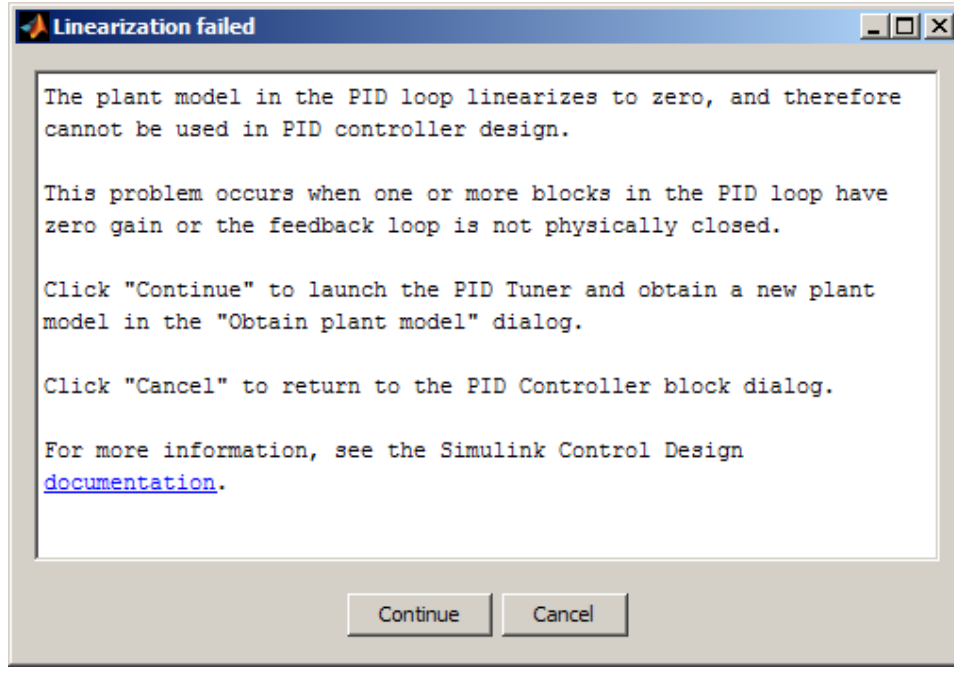
```
% Hide scope
close_system([mdl '/Engine Speed (rpm)'])
% Obtain the linearization input and output points
io = getlinio(mdl);
% Linearize the plant at initial operating point
linsys = linearize(mdl,io)
```

```
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.



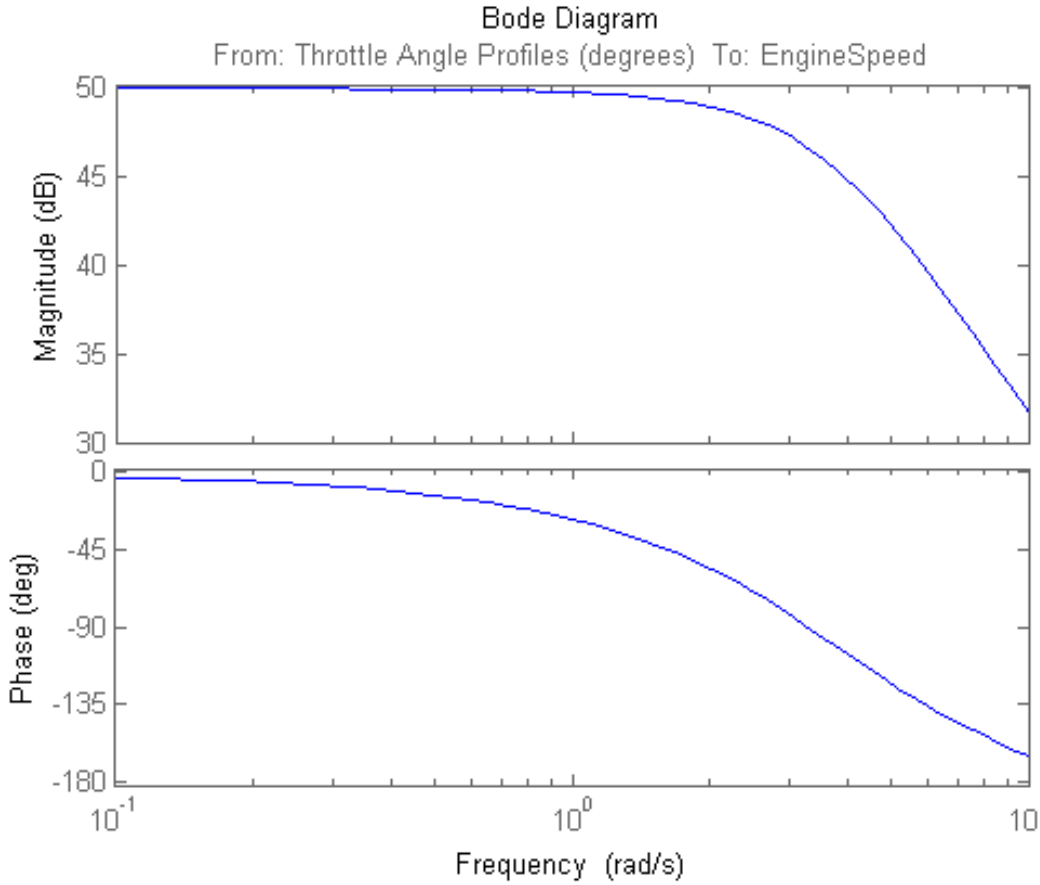
The alternative way to obtain a linear plant model is to directly estimate the frequency response data from the Simulink model, create an FRD system in MATLAB Workspace, and import it back to the PID Tuner to continue PID design.

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.

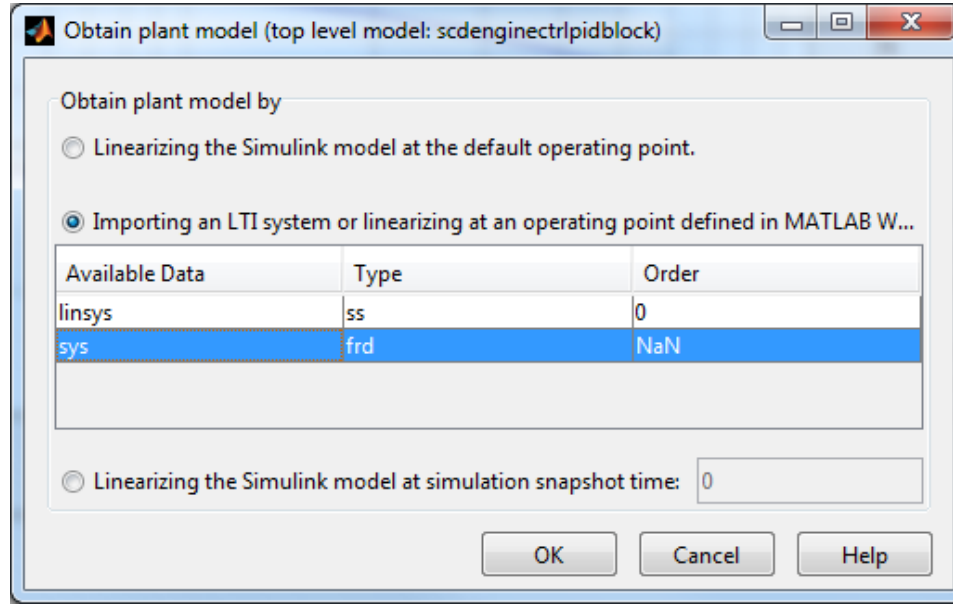
In this example, we create a sine stream that sweeps frequency from 0.1 to 10 rad/sec. Its amplitude is set to be 1e-3. You can inspect the estimation results using the bode plot.

```
% 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 finis
% Display Bode plot
figure;
bode(sys);
```

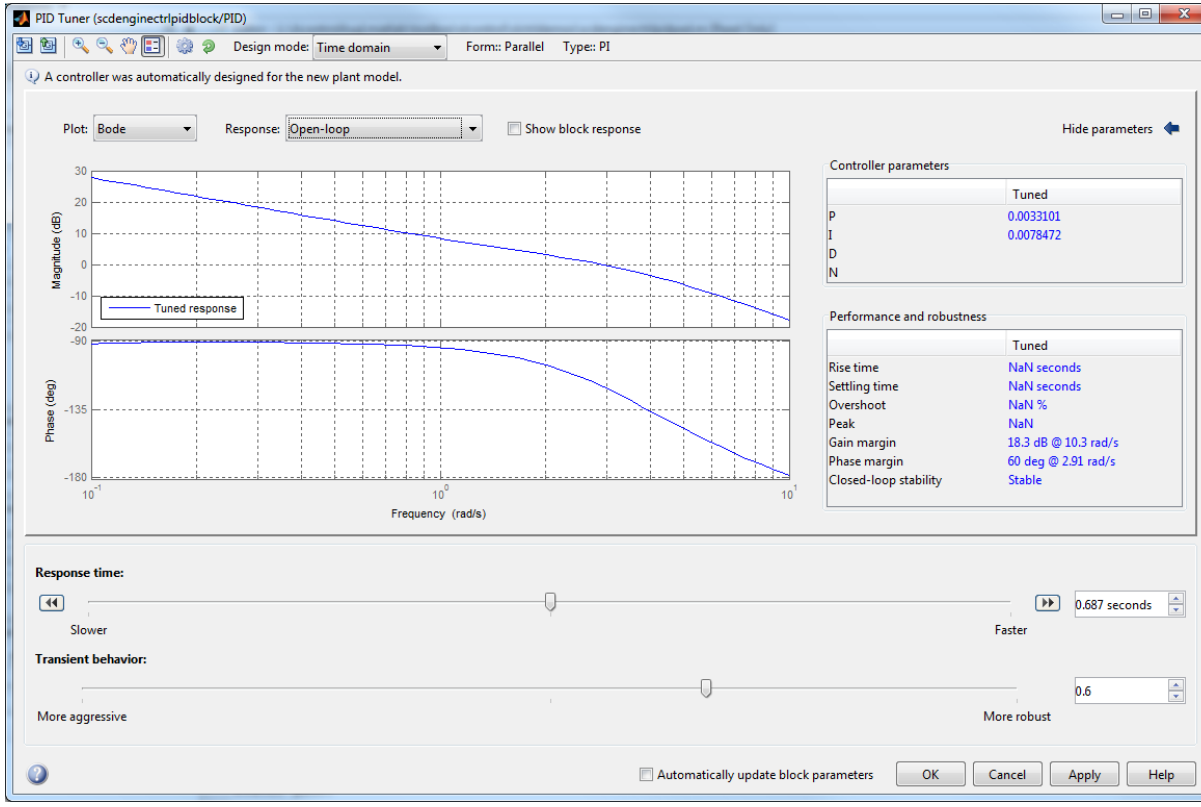


Designing PI with the FRD System in PID Tuner

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. Open the PID block dialog and click Tune button again. When the information dialog shows up, click "Continue" to resume the launching process. After the PID Tuner is launched, an "Obtain Plant Model" dialog shows up and asks you to import a new plant model for PID design.



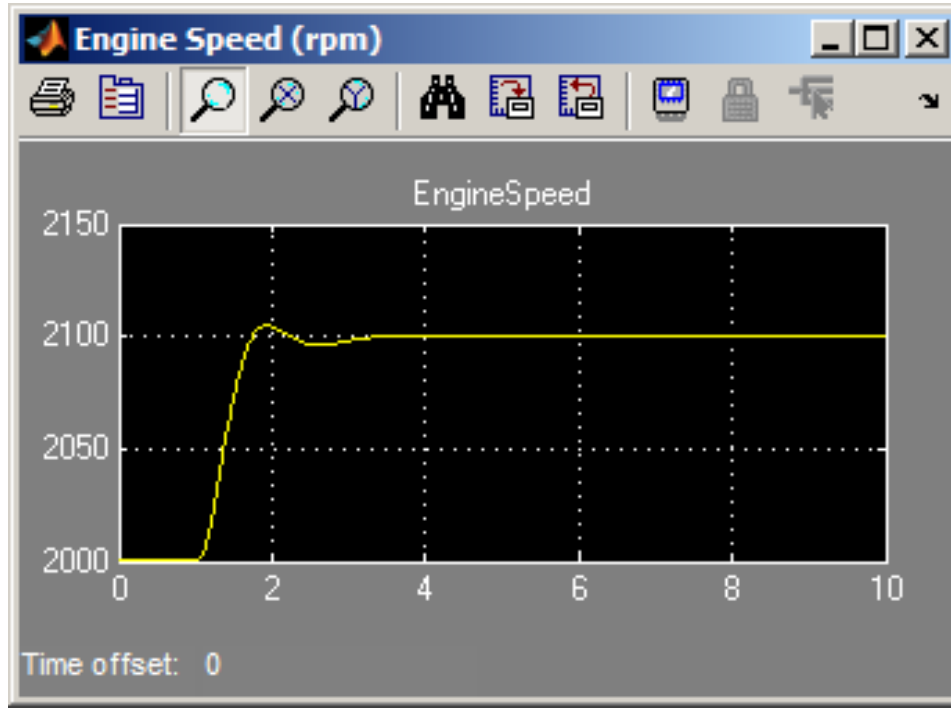
Click the radio button in the middle, select "sys" from the list, and click "OK" to import the FRD system into the PID Tuner. The automated design returns a stabilizing controller. Select "Open-loop" as the response in the Bode plot and the plot shows reasonable gain and phase margin. Time domain response plots are not available for FRD plant models.



Click "OK" button in the PID Tuner. The PID Tuner writes the P and I gains to the PID block and closes.

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

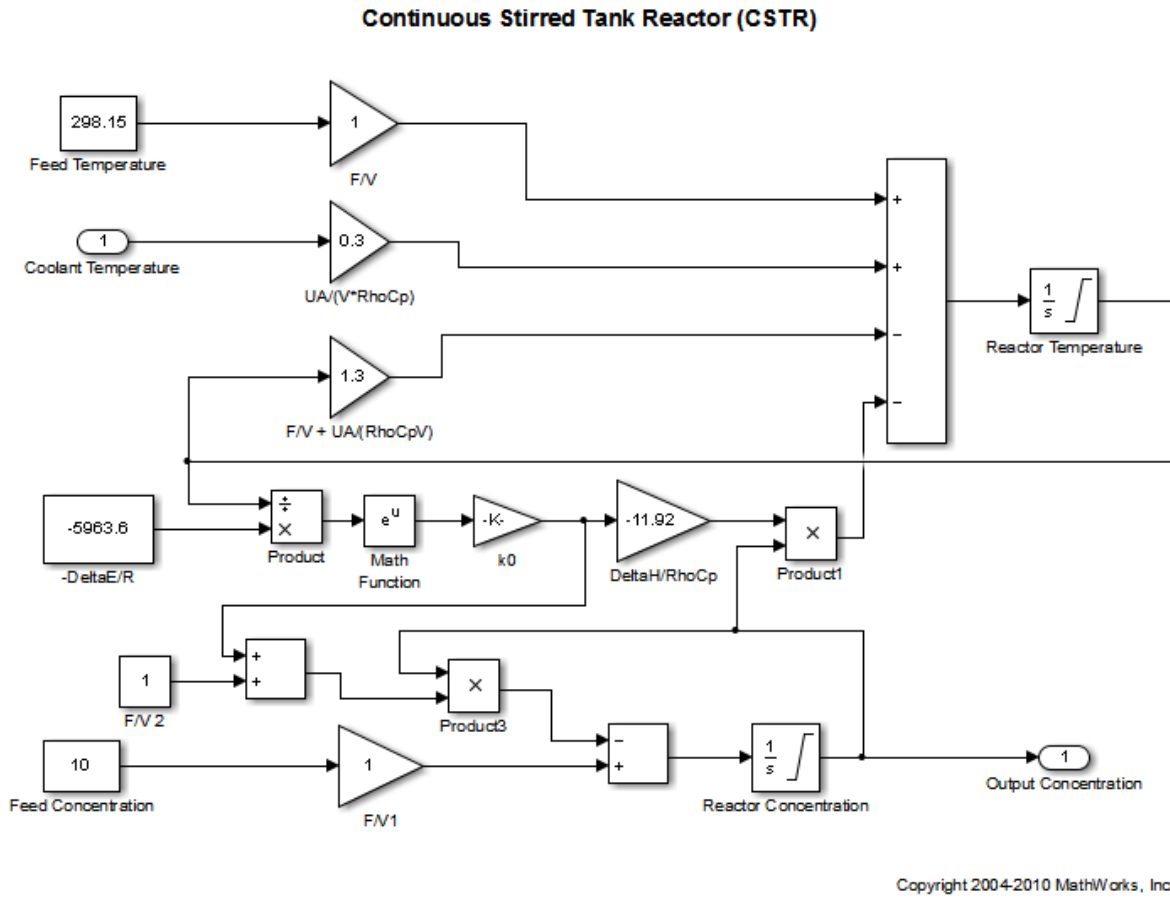
This example shows how to design an array of PID controllers for a nonlinear plant in Simulink that operates over a wide range of operating points.

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 loop, first compute equilibrium operating points with the `findop` command. Then linearize the plant at each operating point with the `linearize` command.

```
% Obtain default operating point
op = operspec(mdl);
% Set the value of output concentration C to be known
op.Outputs.Known = true;
% Specify operating regions
C = [2 3 4 5 6 7 8 9];
% Initialize an array of state space systems
Plants = rss(1,1,1,8);
for ct = 1:length(C)
    % Compute equilibrium operating point corresponding to the value of C
    op.Outputs.y = C(ct);
```

```
    opoint = findop(mdl,op,findopOptions('DisplayReport','off'));
    % Linearize plant at this operating point
    Plants(:,:,ct) = linearize(mdl, opoint);
end
```

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 =

     1     1     0     0     0     0     1     1
```

Designing PID Controllers for the Plant Models

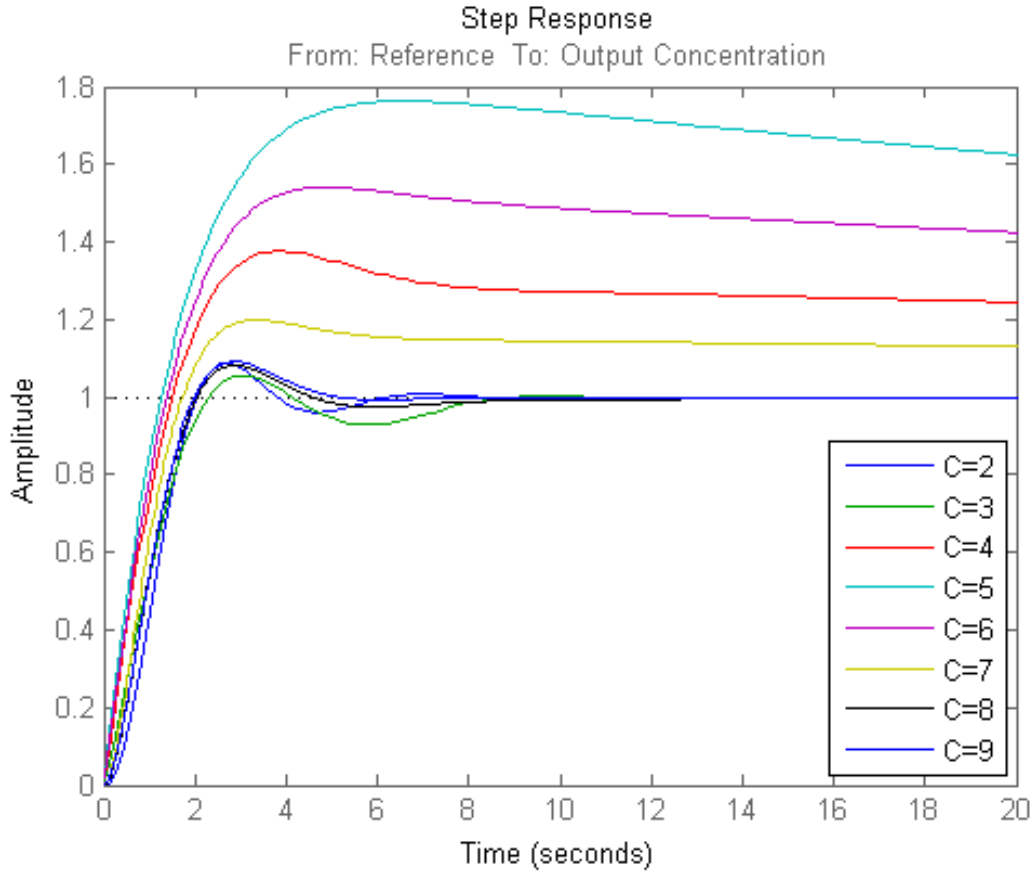
To design multiple PID controllers in batch, we can use the `pidtune` command. The following command will generate an array of PID controllers in parallel form. The desired open loop crossover frequency is at 1 rad/sec and the phase margin is the default value of 60 degrees.

```
% 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 = -12.5, Ki = -0.108, Kd = -14.4, 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);
    set(sys,'Name',['C=',num2str(C(ct))],'InputName','Reference');
    % Plot step response
    stepplot(sys,20);
end
legend('show','location','southeast')
```



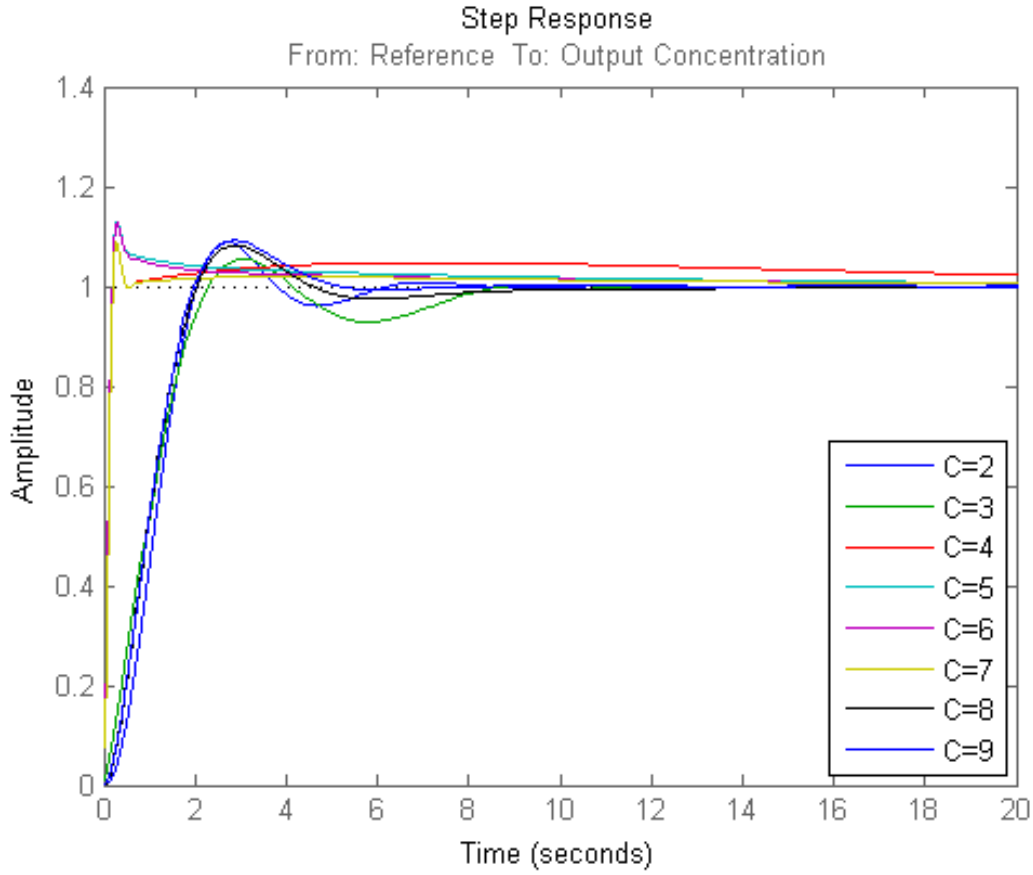
All the closed loops are stable but the overshoots of the loops with unstable plants (C=4, 5, 6 and 7) are too large. It can be improved by increasing the open loop bandwidth to 10 rad/sec.

```
% Design controllers for unstable plant models
Controllers(:,:,3:6) = pidtune(Plants(:,:,3:6),'pidf',10);
% Display controller for C=4
Controllers(:,:,4)
```

```
ans =

             1             s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = -163, Ki = -13.2, Kd = -138, Tf = 0.0471

Continuous-time PIDF controller in parallel form.
```

All the closed loop responses are satisfactory now.

```
% 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')
```



We designed an array of PID controllers and each of them should give reasonable performance around the local operating point. The next step is to implement the scheduling mechanism, which is beyond the scope of this example.

Close the model.

```
bdclose(mdl);
```

Design and Analysis of Control Systems

In this section...
"Compensator Design Process Overview" on page 4-68
"Beginning a Compensator Design Task" on page 4-68
"Selecting Blocks to Tune" on page 4-70
"Selecting Closed-Loop Responses to Design" on page 4-73
"Selecting an Operating Point" on page 4-75
"Creating a SISO Design Task" on page 4-78
"Completing the Design" on page 4-89

Compensator Design Process Overview

Compensator design in the Control and Estimation Tools Manager involves the following steps:

1. "Selecting Blocks to Tune" on page 4-70
2. "Selecting Closed-Loop Responses to Design" on page 4-73
3. "Selecting an Operating Point" on page 4-75
4. "Creating a SISO Design Task" on page 4-78
5. "Completing the Design" on page 4-89

Beginning a Compensator Design Task

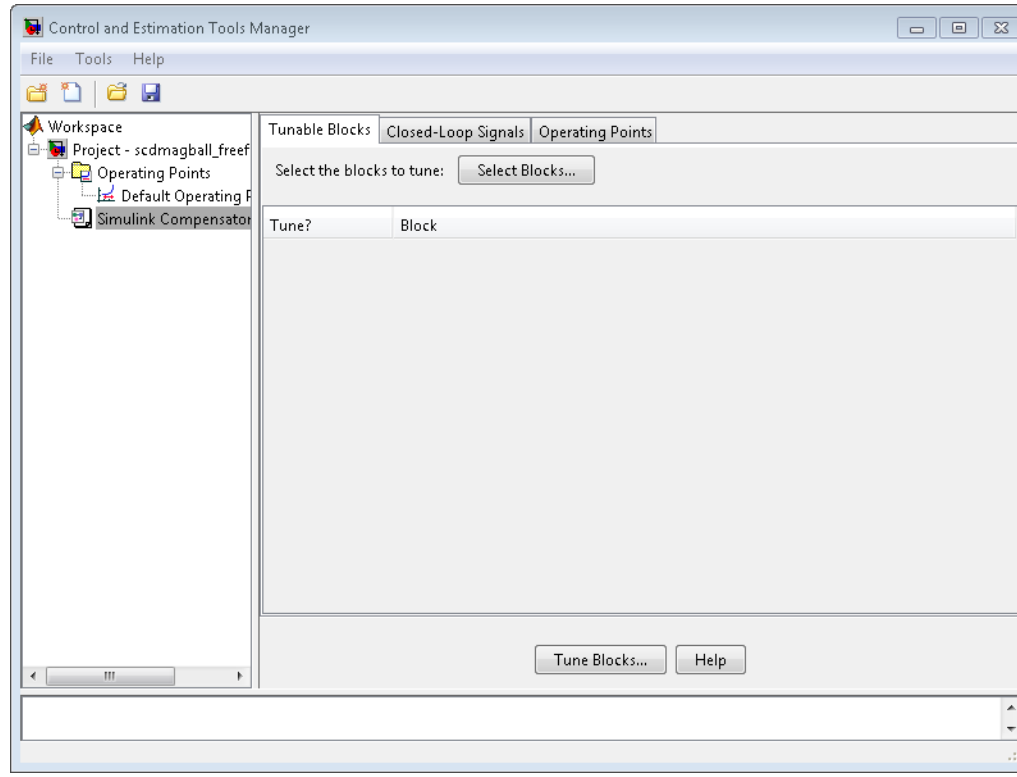
Before you begin this compensator design example, close the Control and Estimation Tools Manager.

To begin a new compensator design task for the `scdmagball_freeform` model:

1. Enter `scdmagball_freeform` at the MATLAB command line to open the `scdmagball_freeform` model.

2 Select **Analysis > Control Design > Compensator Design** from the `scdmagball_freeform` window.

The Control and Estimation Tools Manager opens and creates a new compensator design task, as shown in the following figure.



The project tree in the left pane of the Control and Estimation Tools Manager now shows a **Simulink Compensator Design Task** node as part of **Project - scdmagball_freeform** in addition to the **Operating Points** node. You can select a node within the tree to display its contents in the right pane.

- For information on the **Tunable Blocks** pane within the **Simulink Compensator Design Task** node, refer to “Selecting Blocks to Tune” on page 4-70.

- For information on the **Closed-Loop Signals** pane within the **Simulink Compensator Design Task** node, refer to "Selecting Closed-Loop Responses to Design" on page 4-73.
- For information on the **Operating Points** node or the **Operating Points** pane within the **Simulink Compensator Design Task** node, refer to "Selecting an Operating Point" on page 4-75.

Selecting Blocks to Tune

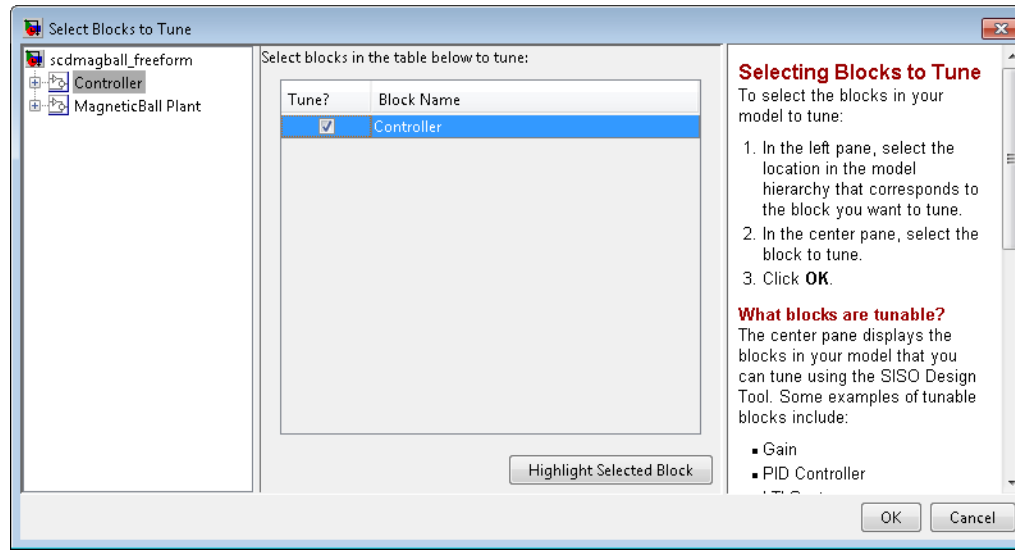
How to Select Blocks to Tune

This section continues the `scdmagball_freeform` example from "Beginning a Compensator Design Task" on page 4-68. At this stage in the example, you have already created a compensator design task.

In this step of the compensator design, you select the blocks in your model to tune from a list of tunable blocks in your model. *Tunable blocks* are blocks that you can tune using the SISO Design Tool to achieve the desired response of your system. Typically, these blocks serve as the compensators in your model.

In this example, you tune the compensator block called Controller inside the Controller subsystem of the `scdmagball_freeform` model. To select this block as the block to tune:

1. Select the **Simulink Compensator Design Task** node.
2. In the **Tunable Blocks** pane, click **Select Blocks**. The Select Blocks to Tune dialog box opens.
3. Select the Controller subsystem in the left pane to display that subsystem's tunable blocks within the center pane. Within the center pane, select the check box next to the Controller block's name.



4 Click **OK** to apply your selections and close the dialog box.

What Blocks Are Tunable?

You can tune parameters in the blocks shown in the following table using Simulink Control Design software. The block input and output signals for tunable blocks must have scalar, double-precision values.

Tunable Blocks	Simulink Library
Gain	Math Operations
LTI System	Control System Toolbox
Discrete Filter	Discrete
PID Controller (one-degree-of-freedom only)	• Continuous • Discrete • Simulink Extras Additional Linear

Tunable Blocks	Simulink Library
State-space blocks	• Continuous • Discrete • Simulink Extras Additional Linear
Zero-pole blocks	• Continuous • Discrete • Simulink Extras Additional Linear
Transfer function blocks	• Continuous • Discrete • Simulink Extras Additional Linear

You can also tune the following versions of the blocks listed in the table:

- 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.

Creating Custom Configuration Functions

When you have masked subsystems that you want to tune in your model, they will not automatically appear in the list of tunable blocks. For them to appear in the list, you need to create a custom configuration function for the masked subsystem. The custom configuration function serves the following functions:

- It informs the Simulink Control Design software that you want this block to be available for tuning.

- It determines how you want the SISO Design Task to treat the block; it describes the relationship between the block mask parameters and the poles and zeros of the transfer function.

To learn how to create a custom configuration function, see Tuning Custom Masked Subsystems.

Selecting Closed-Loop Responses to Design

This section continues the `scdmagball_freeform` example from "Selecting Blocks to Tune" on page 4-70. At this stage in the example a compensator design task has been created, and tunable blocks have been selected.

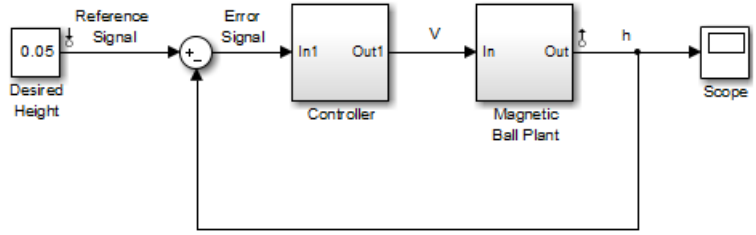
In this step of the compensator design task, you will select the closed loops whose responses you want to design in your model. A closed-loop system is defined by an input point, such as a reference or disturbance signal, and an output point, such as a measured output or actuator signal.

In this example you will design the response of the closed-loop system from the reference signal to the output of the plant model. To set up linear analysis points to define this closed-loop system, perform the following steps:

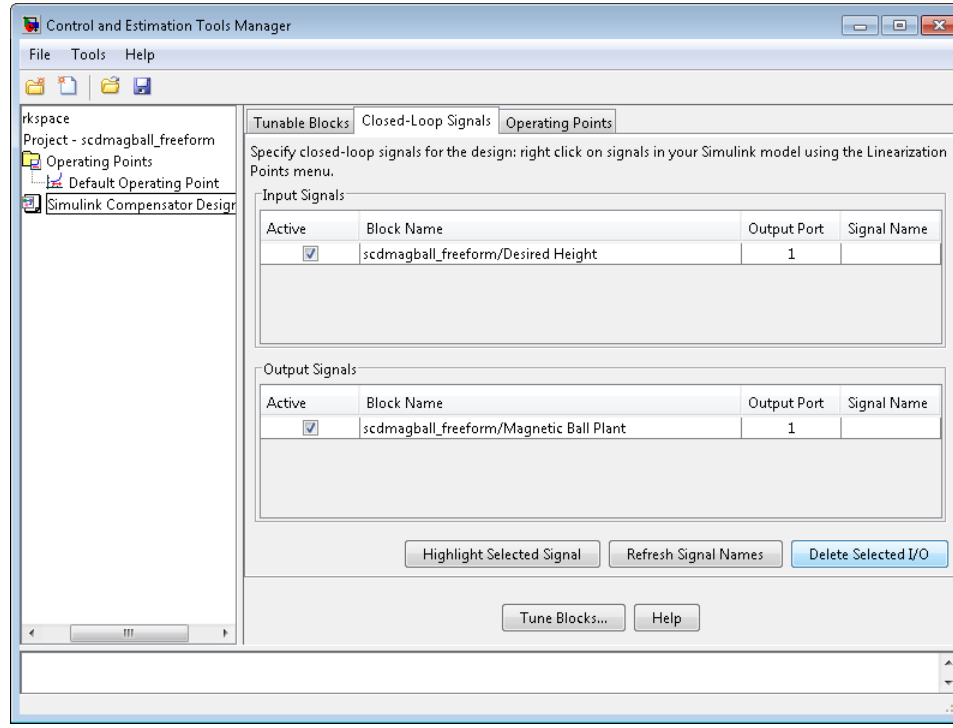
1. On the `scdmagball_freeform` model diagram, position the mouse on the Reference signal between the Desired Height block and the Sum block. Right-click and select **Linear Analysis Points > Input Perturbation** from the menu to add an input point.

2. Position the mouse on the signal line at the output of the Magnetic Ball Plant block. Right-click and select **Linear Analysis Points > Output Measurement** from the menu to add an output measurement point.

The model should now appear as follows:



Within the Control and Estimation Tools Manager, click the **Closed-Loop Signals** tab of the **Simulink Compensator Design Task** node to view the input and output points in the model.



Within this pane you can view the input and output signals in the model and use the **Active** column to select the ones you want to use to define closed-loop systems for compensator design.

Selecting an Operating Point

This section continues the scdmagball_freeform example from “Selecting Closed-Loop Responses to Design” on page 4-73. At this stage in the example, a compensator design task has been created, tunable blocks have been selected, and closed-loop signals have been selected.

In this step of the compensator design task, you will select the operating point that you want to use in the compensator design. The Simulink Control Design software uses the operating point when it linearizes the model before creating a SISO Design Task.

Note A compensator designed for the linearized model is likely to control the behavior of the nonlinear model only in a small region around the operating point that the model was linearized at. Therefore it is important that the linearization of the model is accurate and the selection of the operating point about which the system is linearized is an important step in the compensator design process.

To import an operating point for compensator design, perform the following steps:

1. Select the **Operating Points** node in the Control and Estimation Tools Manager.

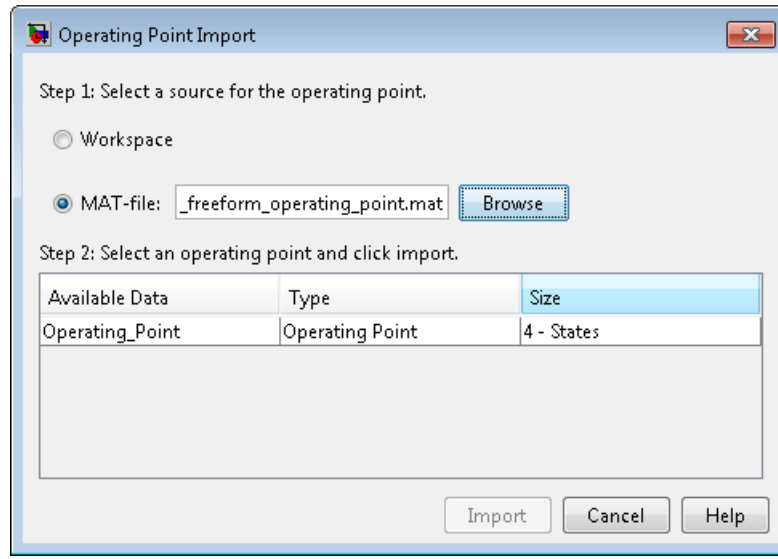
2. Click the **Import** button, in the bottom-right corner of the Control and Estimation Tools Manager.

3. In the Operating Point Import dialog box, select **MAT-file** as the location to import from.

4. Click **Browse** and locate the file `scdmagball_freeform_operating_point.mat` that you previously saved. If you did not previously save an operating point, browse to *`matlabroot`*`/toolbox/slcontrol/slctrldemos/scdmagball_freeform_operating_po`

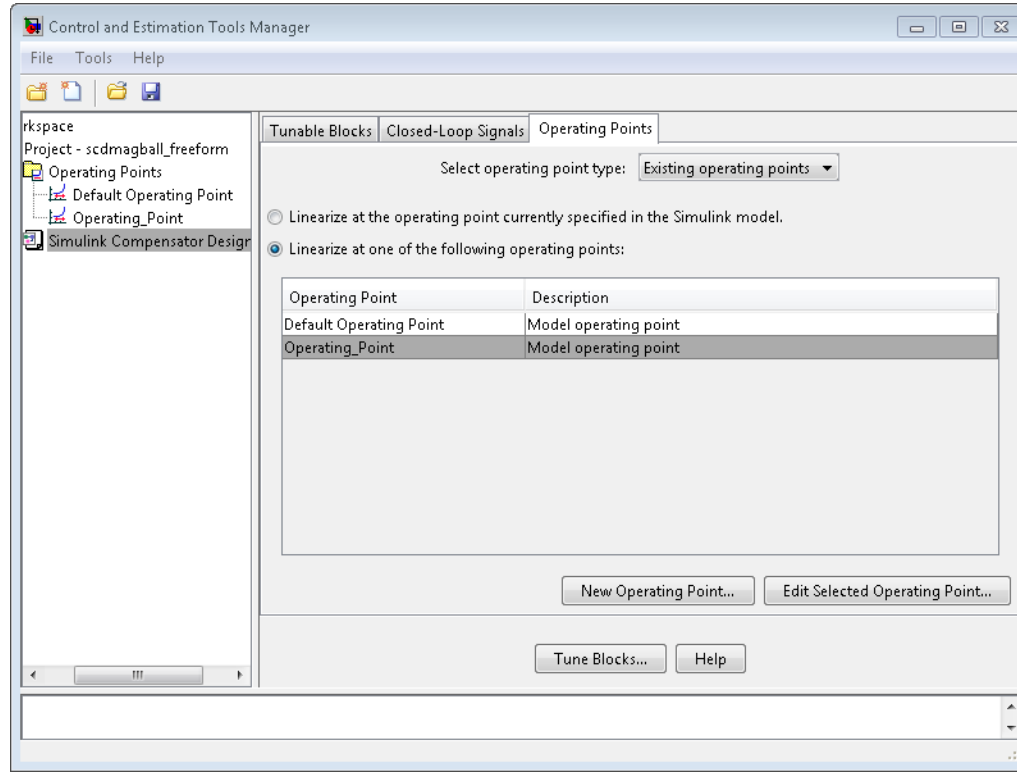
5. Click **Open** to return to the Operating Point Import dialog box.

   The Operating Point Import dialog box now shows all the operating points available within the selected MAT-file. In this case just a single operating point is contained in the MAT-file.



6 Select this operating point and click **Import** to import it into the Control and Estimation Tools Manager.

Click the **Operating Points** tab in the **Simulink Compensator Design Task** node to select an operating point for the compensator design. For this example, you should use the operating point that you just imported, called **Operating_Point**. To specify this operating point, first select the **Linearize at one of the following operating points** option. Then select **Operating_Point** in the list, as shown in the following figure.



Creating a SISO Design Task

- "What is a SISO Design Task?" on page 4-78
- "Configuring Design Plots" on page 4-79
- "Configuring Analysis Plots" on page 4-82
- "Control Design Linearization Options" on page 4-87
- "Designing Compensators for Plants with Time Delays" on page 4-88

What is a SISO Design Task?

This section continues the `scdmagball_freeform` example from "Selecting an Operating Point" on page 4-75. At this stage in the example, a compensator

design task has been created, and tunable blocks, closed-loop signals, and an operating point have been selected.

In this step of the compensator design task, you will create and configure a **SISO Design Task** in the Control and Estimation Tools Manager. The **SISO Design Task** includes several tools for tuning the response of SISO systems:

- A graphical editing environment in the SISO Design Tool window that contains design plots such as root-locus, and Bode diagrams
- An LTI Viewer window where you can view time and frequency analysis plots of the system
- A compensator editor where you can directly edit the block mask parameters or the poles and zeros of compensators in your system
- A tool that automatically generates compensators using PID, internal model control (IMC), or linear-quadratic-Gaussian (LQG) methods (uses the Control System Toolbox software)
- Optimization-based tuning methods that automatically tunes the system to satisfy design requirements (available when you have the Simulink Design Optimization product)

The Design Configuration Wizard guides you through the selection of the open- and closed-loop systems you want to design and the configuration of the design and analysis plots you want to use in the **SISO Design Task**. To launch the wizard, click **Tune Blocks** in the Simulink Compensator Design Task node. The wizard opens in a separate window.

The first page of the wizard provides an overview of the design configuration process and lists some issues to consider when selecting design and analysis plots. Click **Next** to continue to step 1 of the design configuration process on the second page of the wizard.

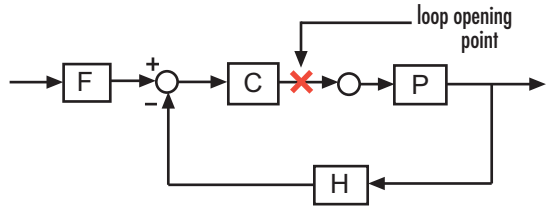
Configuring Design Plots

In step 1, select the open- and closed-loop systems that you want to design in your model, and up to six corresponding design plots you want to use.

Open-loop design allows you to design the response of a closed feedback loop in your model by artificially opening the loop and designing the response of

this *open-loop* system. The open-loop design plots use rules of linear control theory to determine the dynamics of the closed-loop system from those of the open-loop system. Open-loop design is typically used to tune compensators that lie inside feedback loops.

A set of default open-loop systems is created for your model, shown in the lower half of the wizard. To create these open-loop systems, the software artificially opens the feedback loop at the output signal of each tunable block (at the X in the following figure) and unwraps the closed-loop system to create the corresponding open-loop system.



The unwrapped open-loop system, which is -*CPH*, is shown in the following figure. The open-loop design plots show the negative of the unwrapped open-loop, which is *CPH*. This configuration allows you to design controllers using a negative feedback architecture.

Note that elements that are outside the feedback loop, such as the prefilter **F**, are not seen in the open-loop system.

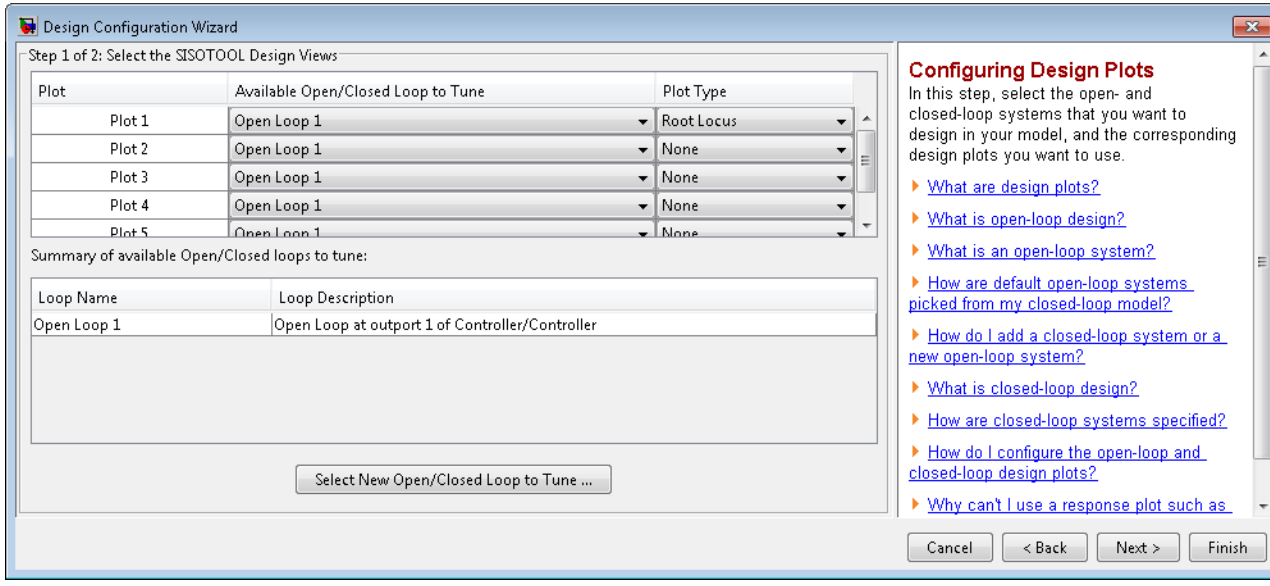
In this example, you will tune the response of **Open Loop 1** which is defined by a loop opening at the output of the Controller block. This open-loop system contains the plant model and the controller. To design this system, select `Open Loop 1` from the menu next to **Plot 1** in the wizard.

Next, select a design plot to use for this open-loop system. Design plots are interactive plots within the SISO Design Tool. You can use them to graphically tune parameters and manually move, add, or remove poles and zeros of the tunable blocks to tune and design the dynamics of open- and closed-loop systems in your model. The following table shows the design plots, along with their uses, available in the SISO Design Tool.

Type of Design Plot	Available Plots in the SISO Design Tool	Use to tune blocks that act as
Open-loop	Root Locus, Nichols, Open-loop bode	Feedback elements
Closed-loop	Closed-loop bode	Feedforward or prefilter elements

You can also use the design plots to specify requirements for stability, performance, or both to use in using optimization-based automated tuning.

For this example, select `Root Locus` from the menu next to **Plot 1** to use this plot type as the design plot for **Open Loop 1**. Step 1 of the wizard should now look similar to the following figure.



Click **Next** to proceed to step 2 of the wizard.

Configuring Analysis Plots

In this step, select the closed-loop responses that you want to view while designing your model, and the corresponding analysis plots you want to use to view them.

Analysis plots are plots that show the responses or dynamics of a closed or open loop systems or tunable blocks in your model. Although you cannot directly edit the analysis plots by graphically moving gains, poles, zeros, etc., changes that you make in the design plots, compensator editor, or automated design tools will affect the responses in the analysis plots. Possible analysis plots include

- Step response
- Impulse response
- Bode and Bode magnitude
- Nyquist
- Nichols
- Pole/Zero

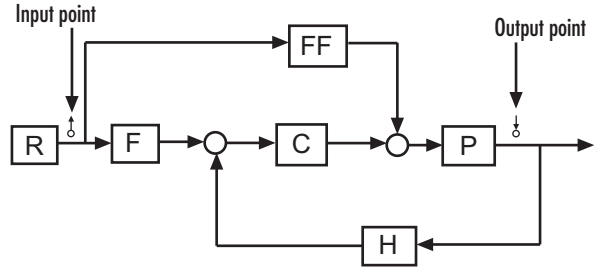
You can use analysis plots to

- Analyze closed-loop, open-loop, and tuned block responses in your control system.
- Define stability and performance requirements for optimization-based automated tuning.

For this example, select `Step` from the menu for **Plot 1** to create a step response analysis plot.

Next, select the closed-loop system that you want to display in this plot. A closed-loop system is a system that has not had any feedback loops opened for open-loop design. It typically defines the system whose response you want to control and it lies between the input and output signals of interest, for example between a reference signal and the plant output signal.

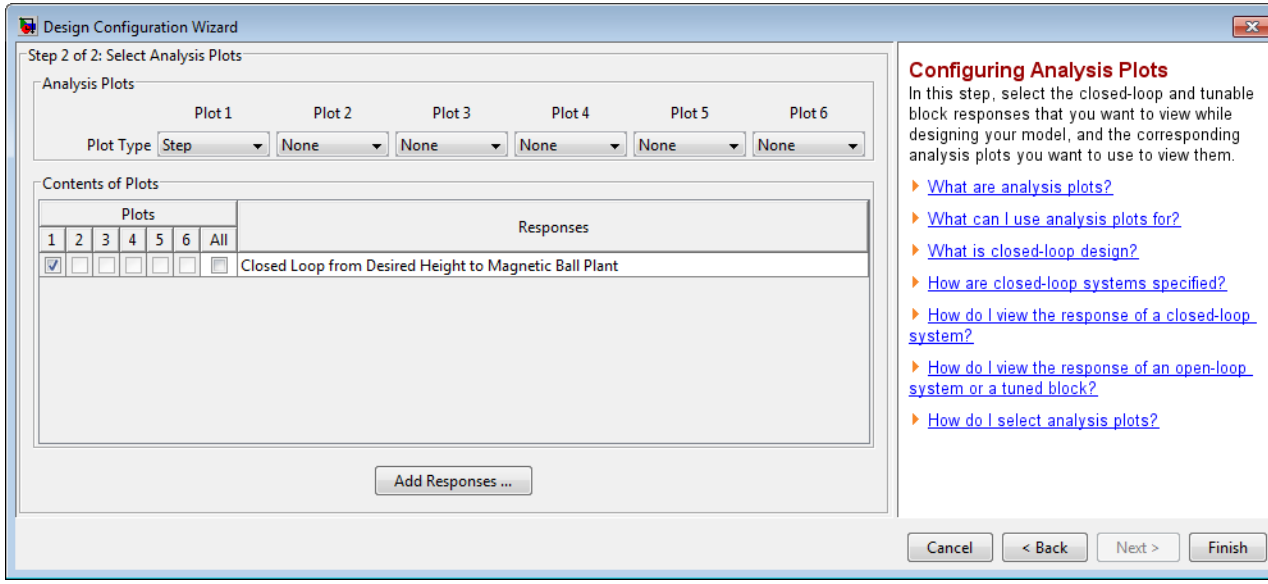
Linearization input and output points placed on signal lines in your model define closed-loop systems. The closed-loop system includes all blocks in the path between the input and the output.



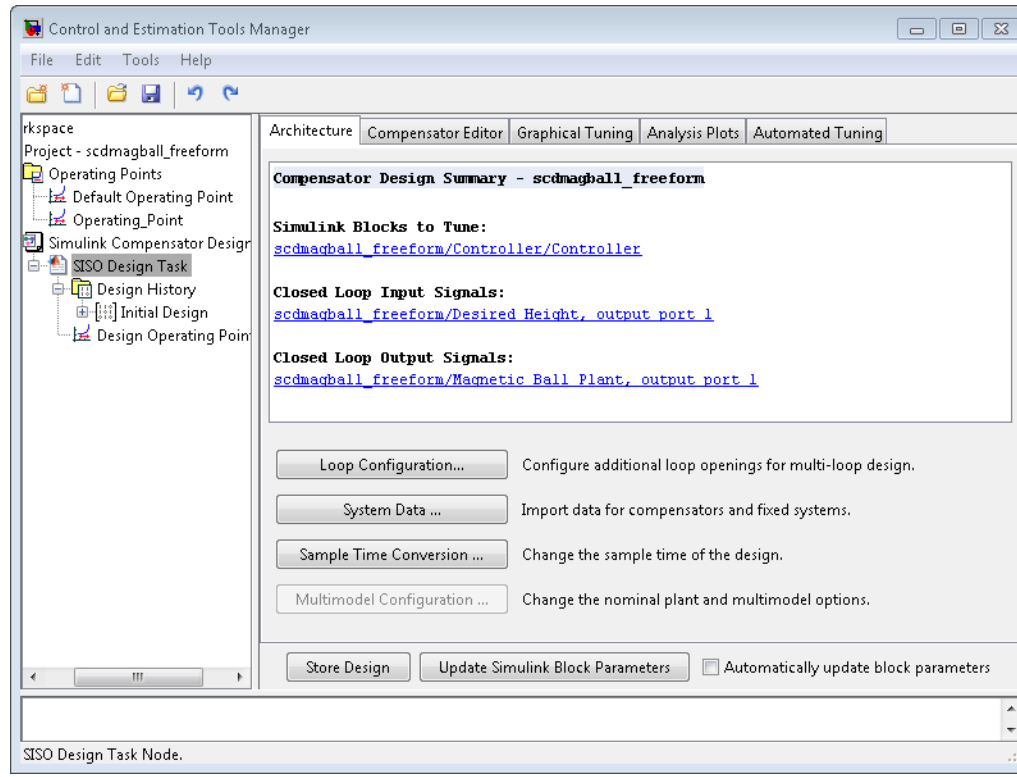
Closed-loop control system

The software automatically displays a list of up to four closed-loop systems in your model, based on the input and output points on the signal lines. In this example, only one closed-loop system appears in the wizard, the closed-loop from the Desired Height signal to the output of the Magnetic Ball Plant Model, because the system only has one input and one output point. You can add additional closed-loop responses, as well as open-loop and tunable block responses. To add a new response, click the **Add Responses** button and complete the Select a New Response to Analyze dialog box.

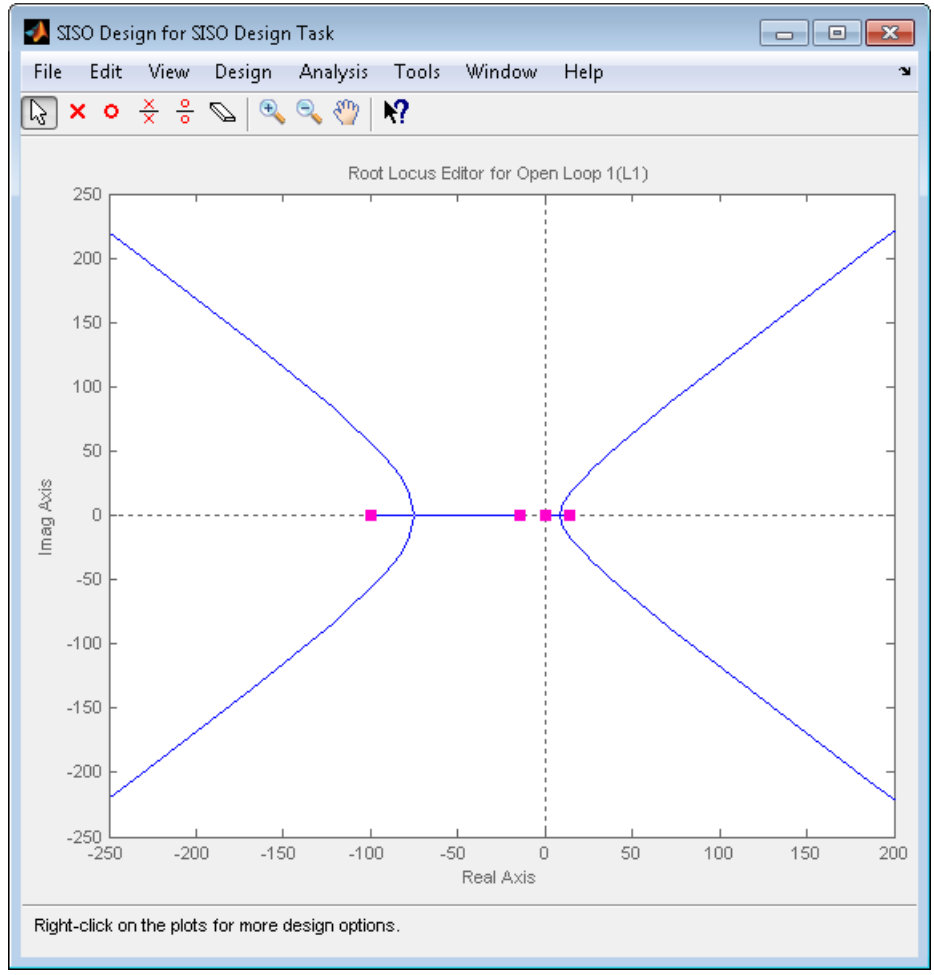
To display the current closed-loop system in the step response plot of **Plot 1**, select the check box under **Plot 1** to the left of the closed-loop system. Step 2 of the wizard should now look similar to the following figure.



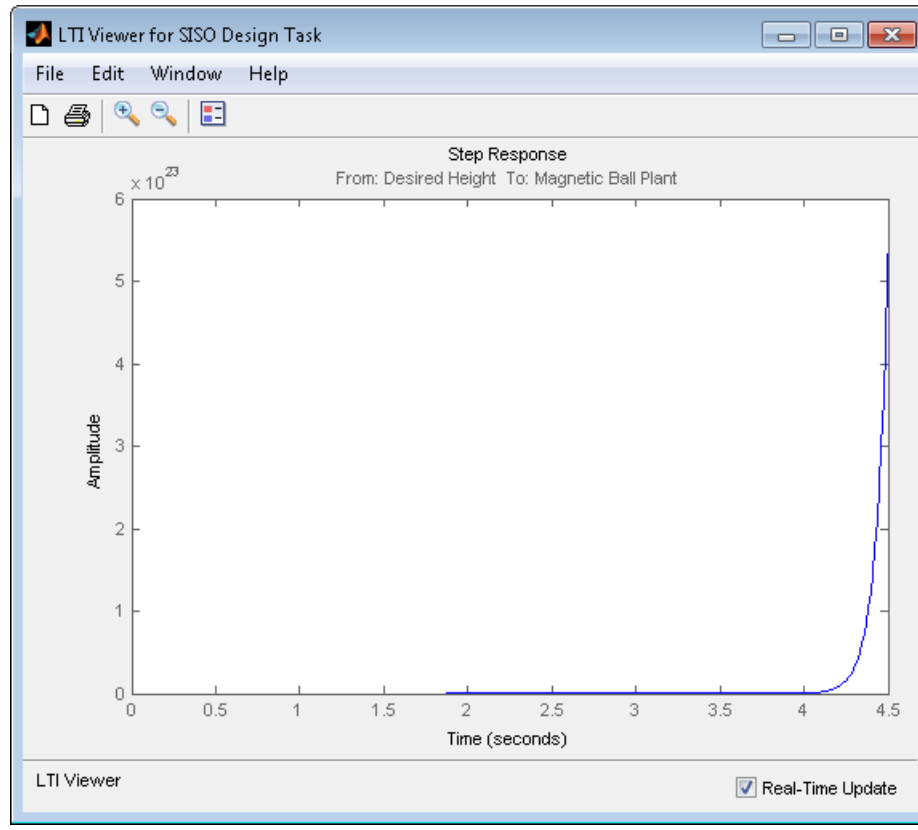
Click **Finish** to complete the wizard and create the **SISO Design Task** underneath the **Simulink Compensator Design Task** node within the Control and Estimation Tools Manager, as shown in the following figure.



The **SISO Design Task** also includes the design plots you configured in the Design Configuration Wizard. They appear within the SISO Design Tool window, as shown in the following figure.

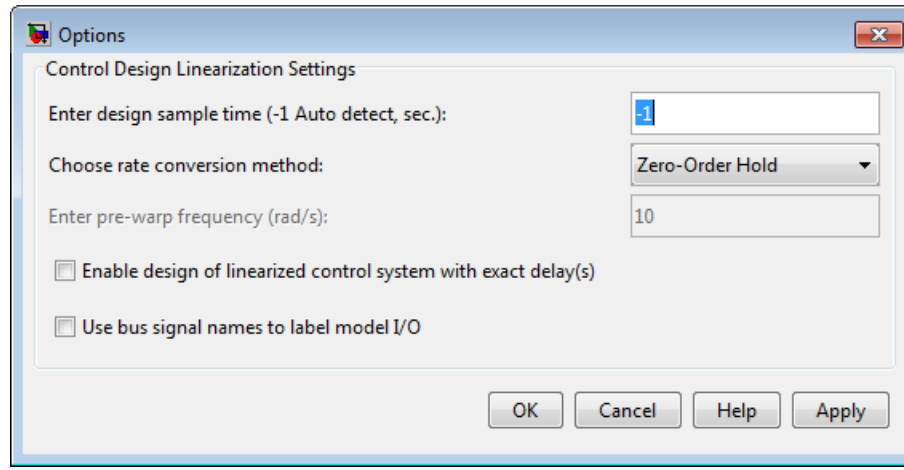


In addition, the **SISO Design Task** also includes the analysis plots you configured in the Design Configuration Wizard. They appear within the LTI Viewer window, as shown in the following figure.



Control Design Linearization Options

To modify or adjust the settings used to linearize a model when creating a SISO Design Task, click the **Simulink Compensator Design Task** node, and then select **Tools > Options**. The Options dialog box opens.



Specify the linearization sample time and rate conversion method. If, for the **Rate conversion method** parameter, you specify `Tustin W/Prewarping`, you must also specify a pre-warp frequency.

Designing Compensators for Plants with Time Delays

You can design compensators for plants with time delays using the tools in the SISO Design Task. These tools automatically create a linear model of your plant. Within this model, you can represent time delays in two ways—using Padé approximation or exact delay.

To represent time delays in the linear plant model using...	You must...
Padé approximation representations	Specify the Padé order in the Block Parameters window for each Simulink blocks with delays.
Exact delay representations	Open the Simulink Compensator Design Task, and select **Tools > Options**. Then, in the Options dialog box, select **Enable design of linearized control systems with exact delay(s)**.

Note Some tools do not support exact time delays and automatically compute a Padé approximation for delays. In this case, you receive a notification. The software uses the Padé order specified in SISO Tool Preferences and ignores the Padé order specified in your block. For more information, see "Time Delays Pane".

For more information on the linearizing models with time delays, see "Models with Time Delays" on page 2-127. For more information on the tools available for compensator design, see "Tools for Compensator Design" on page 4-89.

Completing the Design

- "Tools for Compensator Design" on page 4-89
- "Storing and Retrieving Designs" on page 4-94
- "Writing the Design to the Simulink Model" on page 4-96
- "Compare and Contrast the SISO Design Task and Enhanced SISO Design Task" on page 4-98
- "Design Operating Point Node" on page 4-101
- "SISO Tool Options" on page 4-102

Tools for Compensator Design

This section continues the scdmagball_freeform example from "Creating a SISO Design Task" on page 4-78. At this stage in the example, a compensator design task has been created, tunable blocks, closed-loop signals, and an operating point have been selected, design and analysis plots have been created, and a **SISO Design Task** node has been created in the Control and Estimation Tools Manager.

In this step of the compensator design task, you will complete the design of the compensator in the scdmagball_freeform model, using the **SISO Design Task** node. For a more detailed discussion of the **SISO Design Task** node, refer to the Control System Toolbox documentation.

The **SISO Design Task** node contains five panes with various tools for designing the compensators in your system.

- **Architecture**:
  - Configure loops for multi-loop design by opening signals to remove the effects of other feedback loops.
  - Import compensators into your system.
  - Convert the sample time of the system or switch between different sample times to design different compensators.
- **Compensator Editor**:
  - Directly edit the poles, zeros, and gains of the compensator.
  - Add or remove poles and zeros to the compensators.
- **Graphical Tuning**:
  - Configure design plots in the SISO Design Tool.
  - Use design plots to graphically manipulate the response of the system.
- **Analysis Plots**:
  - Configure analysis plots in the LTI Viewer.
  - Use analysis plots to view the response of open- or closed-loop systems.
- **Automated Tuning**: Design compensators using one of several automated methods.
  - Automatically generate compensators using PID, internal model control (IMC), or linear-quadratic-Gaussian (LQG) methods (uses Control System Toolbox software).
  - Use optimization-based methods that automatically tune the system to satisfy design requirements (available when you have the Simulink Design Optimization product).

You can use any of these design methods, or a combination of methods, to design the compensators for your system. A suitable final design for the Controller of the `scdmagball_freeform` model is:

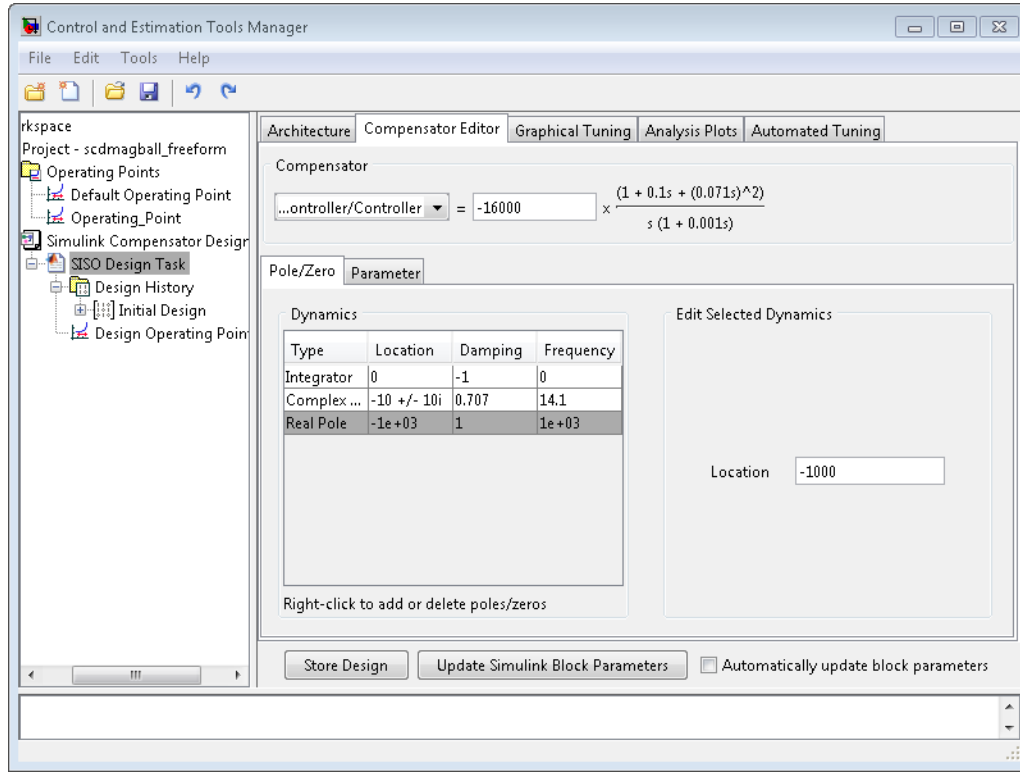
- Gain: -16000

- Integrator at the origin
- Complex zeros at -10±10i
- Real pole at -1000

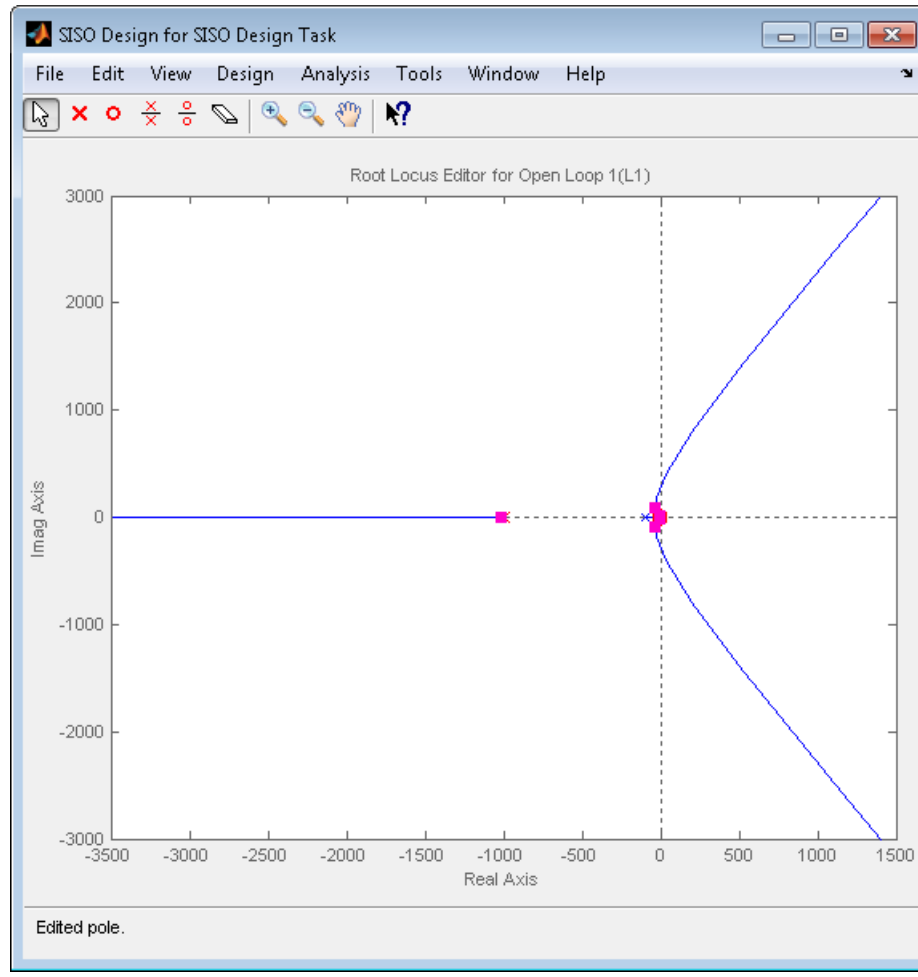
You can use the **Compensator Editor** in the **SISO Design Task** node to specify these settings. The initial design contains an integrator at the origin. Specify the remaining settings as follows:

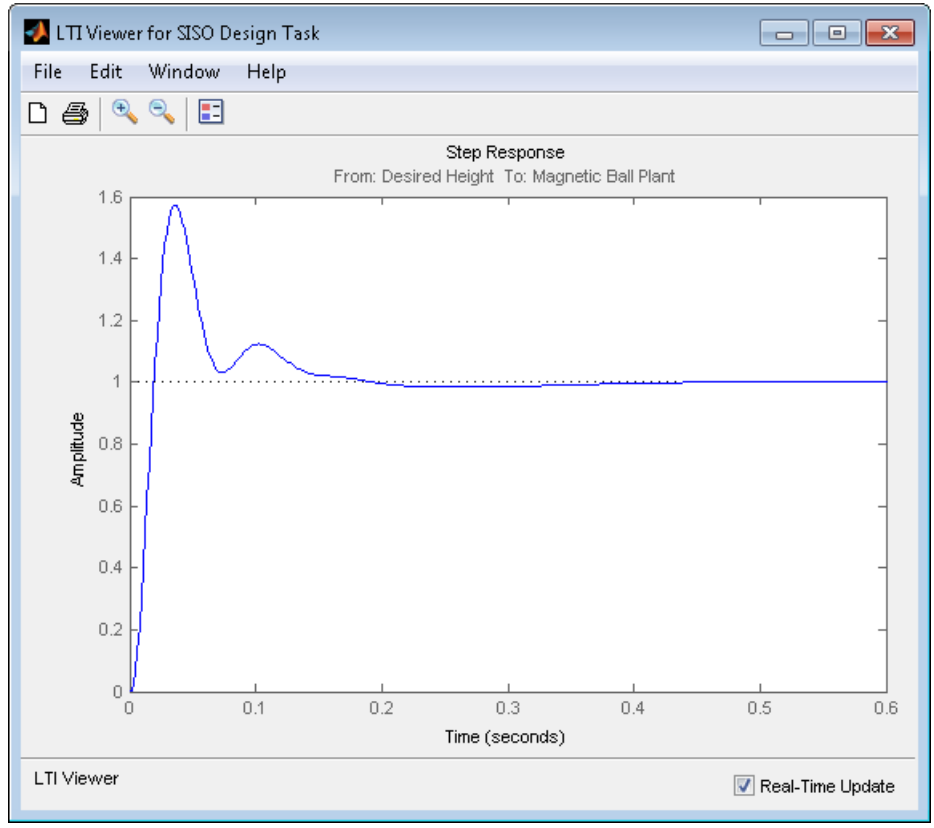
- Gain — Enter `-16000` in the text box to the right of the equal sign in the **Compensator** area.
- Complex zeros — In the **Dynamics** table, right-click and then select **Add Pole/Zero > Complex Zero**. Select the new complex zero in the **Dynamics** table. In the **Edit Selected Dynamics** table:
  - Enter `-10` in the **Real Part** field.
  - Enter `10` in the **Imaginary Part** field.
- Real pole — In the **Dynamics** table, right-click and then select **Add Pole/Zero > Real Pole**. Select the new real pole in the **Dynamics** table. In the **Edit Selected Dynamics** table:
  - Enter `-1000` in the **Location** field.

The Control and Estimation Tools Manager should now appear similar to the following:



With these settings, the root-locus diagram and step-response plot should look similar to the following figures.


SISO Design for SISO Design Task
File
Edit
View
Design
Analysis
Tools
Window
Help
Root Locus Editor for Open Loop 1(L1)
Imag Axis
Real Axis
3000
2000
1000
0
-1000
-2000
-3000
-3500
-3000
-2500
-2000
-1500
-1000
-500
0
500
1000
1500
Edited pole.



Storing and Retrieving Designs

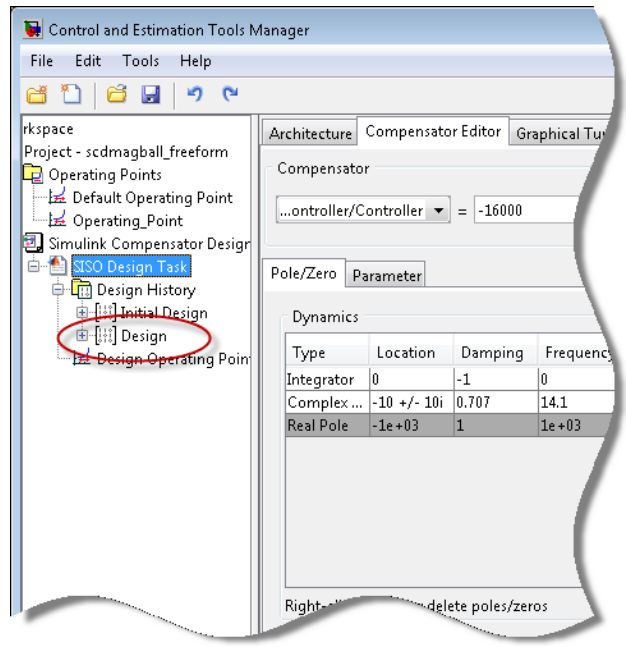
When you design a compensator within a **Simulink Compensator Design Task**, you can store the current design. You can retrieve the stored design at any time. This storage and retrieval capability lets you continue designing and still be able to return to a previously saved version of the design.

This section continues the example from “Completing the Design” on page 4-89. At this stage in the example, a compensator has been designed to control the system. To store the design within the **SISO Design Task** node, perform the following steps:

1 Select the **SISO Design Task** node in the Control and Estimation Tools Manager.

2 Underneath the **SISO Design Task** panes, click the **Store Design** button.

Clicking this button adds the current design to the **Design History** node as shown in the following figure. The default name for the design is **Design**.



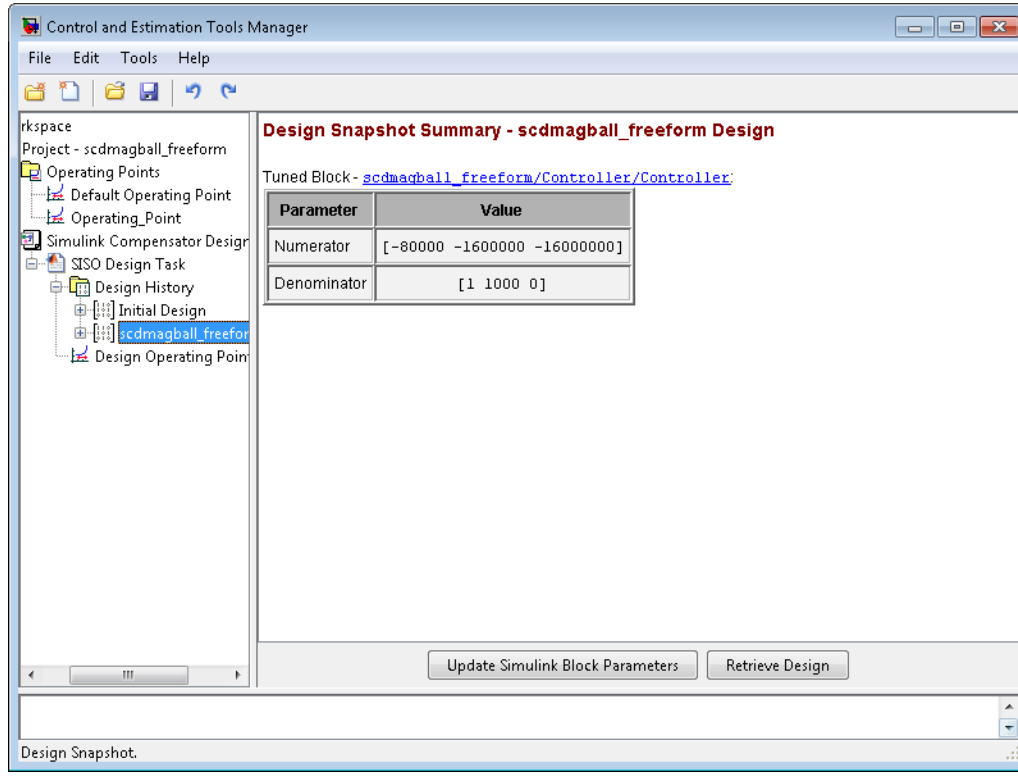
To rename the design to something more descriptive:

1 Right-click the **Design** node underneath the **Design History** node.

2 Select **Rename** from the right-click menu.

3 Enter a name for your design. For this example, call the design **scdmagball_freeform Design**.

The Control and Estimation Tools Manager should now appear as follows:



Note After you store a compensator design, you can continue to refine it. If you make undesired changes, you can retrieve the stored design by selecting it in the **Design History** node and then clicking the **Retrieve Design** button.

Writing the Design to the Simulink Model

When designing a compensator within a **Simulink Compensator Design Task** node, you can write the compensator design to the Simulink model. This is useful when

- You want to see how the current design performs in the full nonlinear model.

- You have completed the design and you want to update the model with the newly designed parameters.

When you write the compensator design to your Simulink model:

- If the block parameters are numerical, the software writes new numerical values to the tuned block.
- If the block parameters are variables in the base workspace or the model workspace (including `Simulink.Parameter` objects), the software writes the tuned values to those variables. The block parameters remain the workspace variables.

There are two ways to write the design to your Simulink model:

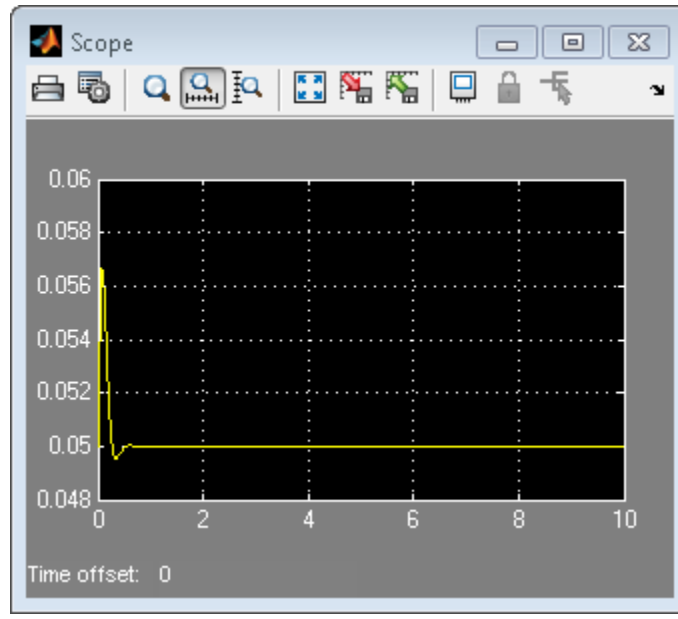
- Write the tuned parameters to your model when you have finished your design — Click the **Design** node of the Control and Estimation Tools Manager and click the **Update Simulink Block Parameters** button.
- Automatically update the block parameters as you tune the design — select the **Design History** node in the Control and Estimation Tools Manager and click the checkbox next to **Automatically update block parameters**.

For example, continue the example from "Storing and Retrieving Designs" on page 4-94. At this stage in the example you have designed a compensator to control the system and stored the design within the **SISO Design Task** node. To write the stored design to the `scdmagball_freeform` model, perform the following steps:

1. Select the **scdmagball_freeform Design** node under the **Design History** node in the Control and Estimation Tools Manager.
2. Click the **Update Simulink Block Parameters** button.

You can now simulate the `scdmagball_freeform` model containing the newly designed Controller block.

After simulation, the Scope block of the `scdmagball_freeform` model should look similar to the following figure.

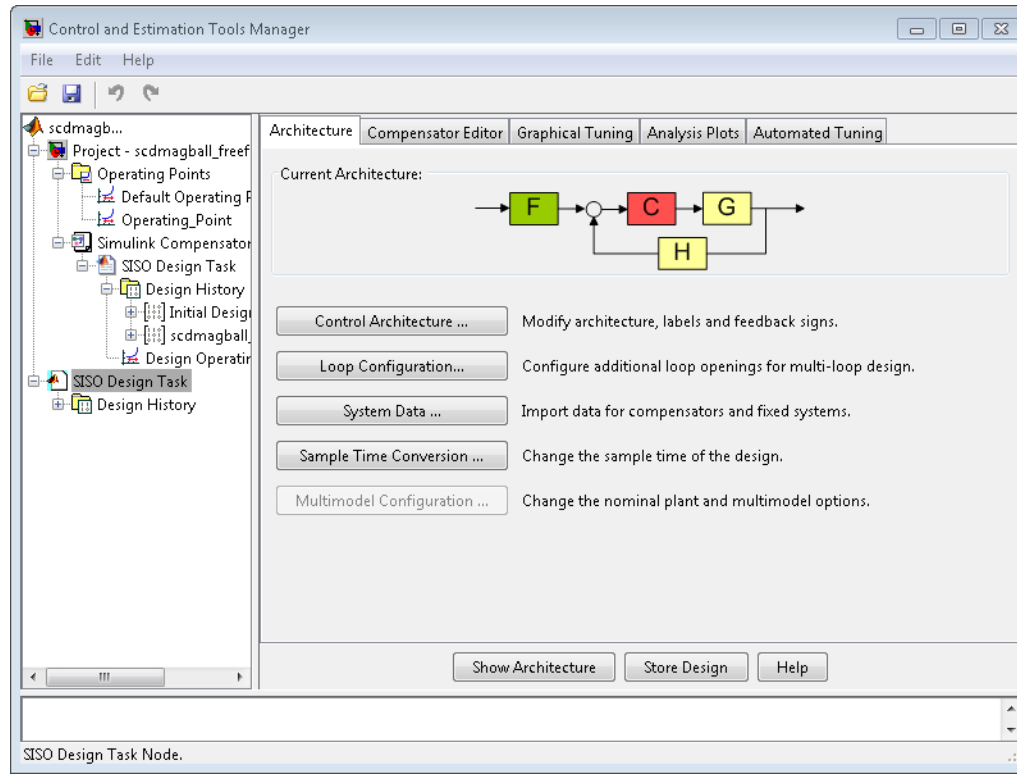


The system is now stable and the height of the magnetic ball settles at the desired height of 0.05 m.

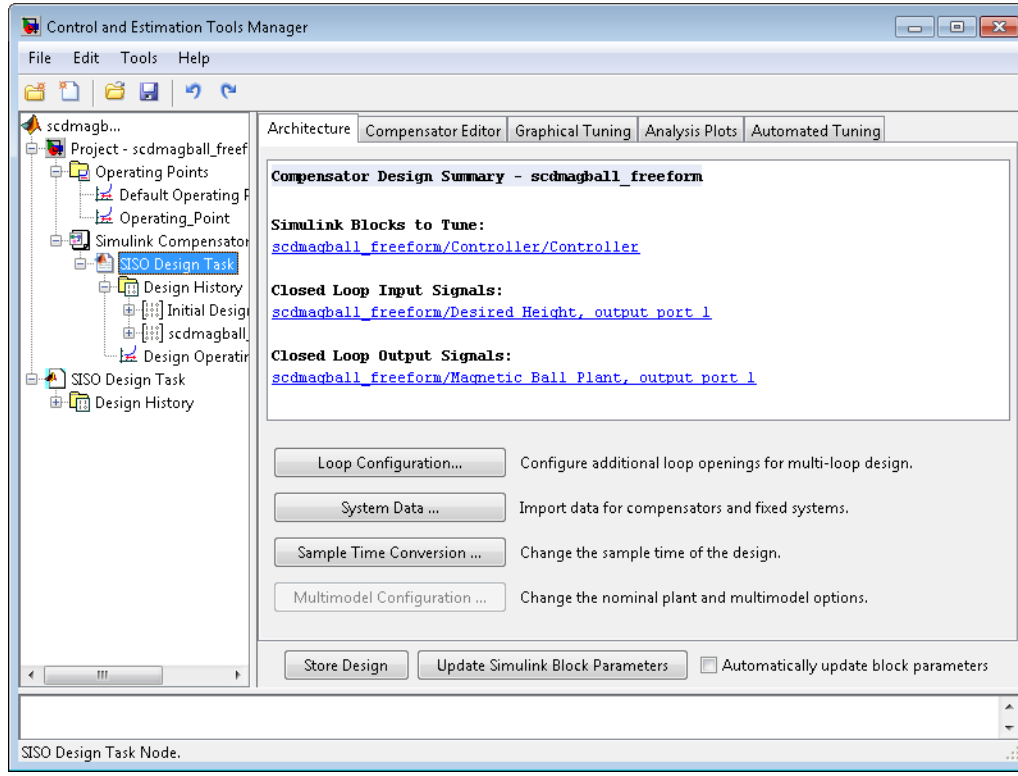
Compare and Contrast the SISO Design Task and Enhanced SISO Design Task

The SISO Design Task is a graphical user interface (GUI) that simplifies the task of designing controllers. This section describes the similarities and differences between the SISO Design Task, which is available in the Control System Toolbox product, and the enhanced SISO Design Task, which is available with the Simulink Control Design product.

The following figure shows the SISO Design Task as it appears in the Control and Estimation Tools Manager.



The following figure shows the enhanced SISO Design Task as it appears under the **Simulink Compensator Design Task** node in the Control and Estimation Tools Manager.



The following table summarizes the similarities and differences between the SISO Design Task and the enhanced SISO Design Task:

Similarities	Differences
• Similar layout • Graphical Tuning, Analysis Plots, and Automated Tuning panes have the same functionality. For more information about these tabs, see “Tools for Compensator Design” on page 4-89.	• Architecture tab — The SISO Tool lets you change the architecture of your system. In contrast, once you create a SISO Design Task you cannot add or delete blocks from your model. Also, the Architecture tab in the **SISO Design Task** node summarizes the Simulink Blocks to Tune, Closed Loop Input

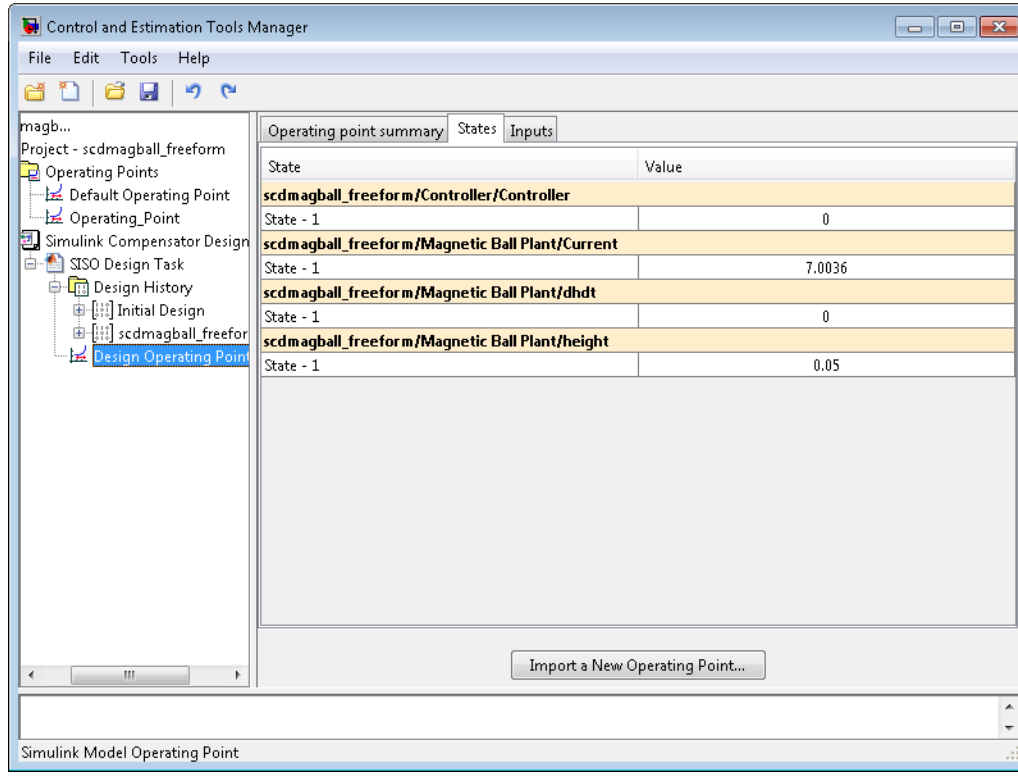
Similarities	Differences
	Signals, and Closed Loop Output Signals. • Compensator Editor tab — The SISO Design Tool lets you tune the poles and zeros of your system. The enhanced SISO Design Tool lets you tune the poles, zeros, and parameters of your system. For more information, see Tuning Simulink Blocks in the Compensator Editor. • When you are satisfied with your system's performance, the enhanced SISO Design Tool lets you click **Update Simulink Block Parameters** to write the parameters back to your Simulink model.

For additional information, see:

- "Creating a SISO Design Task" on page 4-78
- "SISO Design Tool" in the Control System Toolbox documentation

Design Operating Point Node

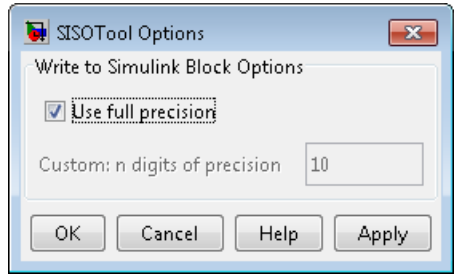
The **Design Operating Point** node is located inside the **Design History** node of the Control and Estimation Tools Manager.



The **States** pane describes the operating point the GUI used to linearize the model. When creating the **SISO Design Task** node, you can use this pane to import a different operating point and to populate the **SISO Design Task** node with a linear model evaluated at the new operating point.

SISO Tool Options

To modify the precision of the numbers calculated by SISO Tool, click the **SISO Design Task** node, and then select **Tools > Options**. The SISOTool Options dialog box opens.



If you select the **Use full precision** check box, the SISO Tool uses the full double-precision data type when writing back to the Simulink block dialog box. If you clear this check box, use **Custom: n digits of precision** to specify the precision you want.

For additional information, see “Creating a SISO Design Task” on page 4-78.

5

Model Verification

- "Monitoring Linear System Characteristics in Simulink Models" on page 5-2
- "Defining a Linear System for Model Verification Blocks" on page 5-4
- "Verifiable Linear System Characteristics" on page 5-5
- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25

Monitoring 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 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 “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

If you have Simulink Verification and Validation™ software, 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.

Defining a 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

  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

  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:

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15

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.

Model Verification 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
   ```

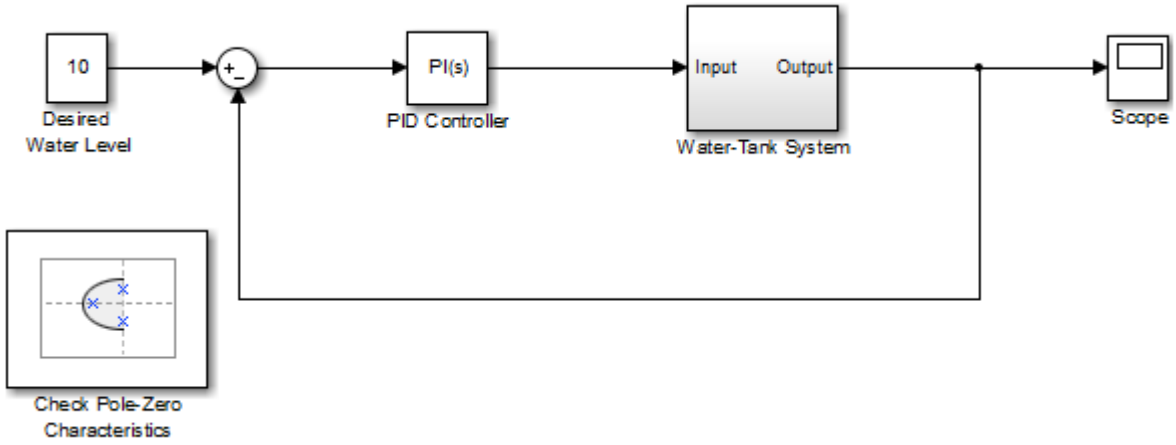
2. Open the Simulink Library Browser by selecting **View > Library Browser** in the Simulink Editor.

3. Add a model verification block to the Simulink model.

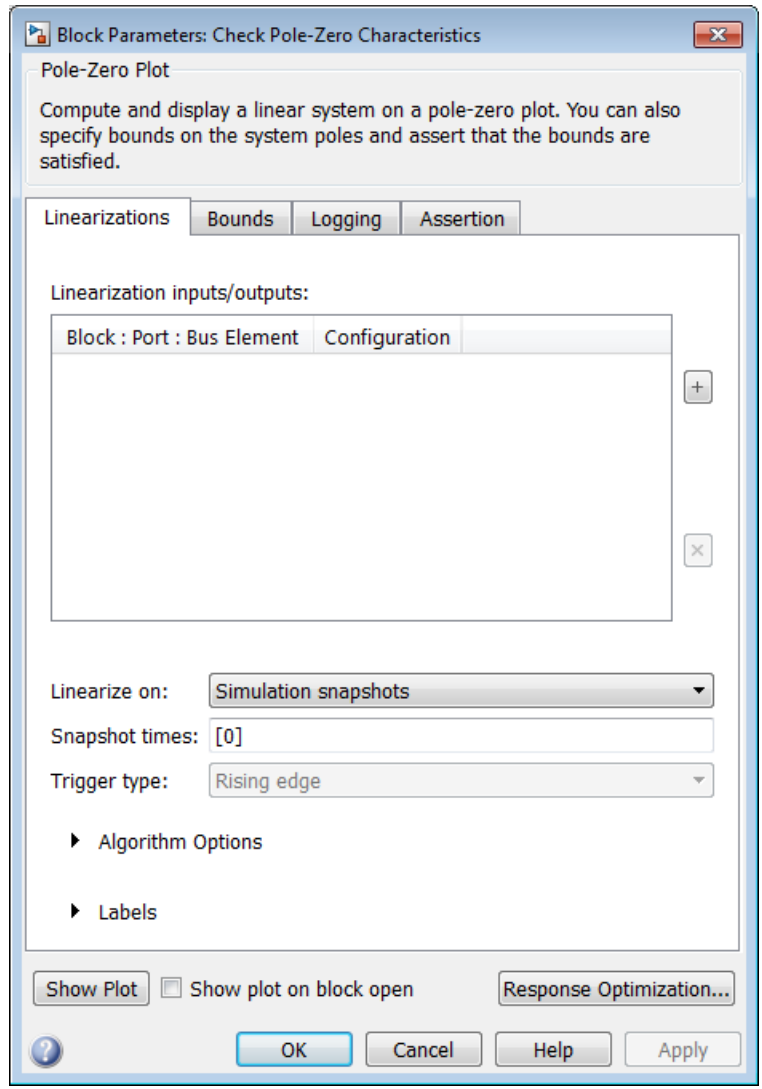
   a. In the **Simulink Control Design** library, select **Model Verification**.

   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.



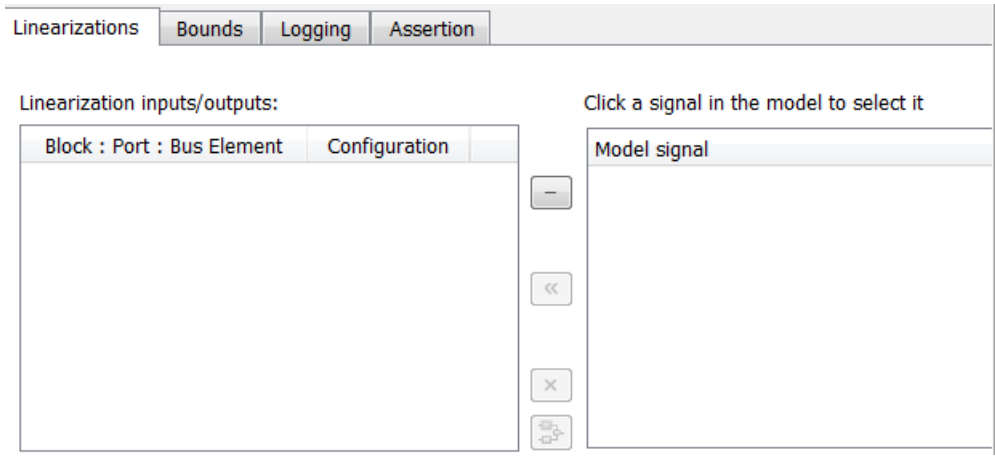
To learn more about the block parameters, see the block reference pages.

5 Specify the linearization input and output to compute the closed-loop poles and zeros.

a To specify an input:

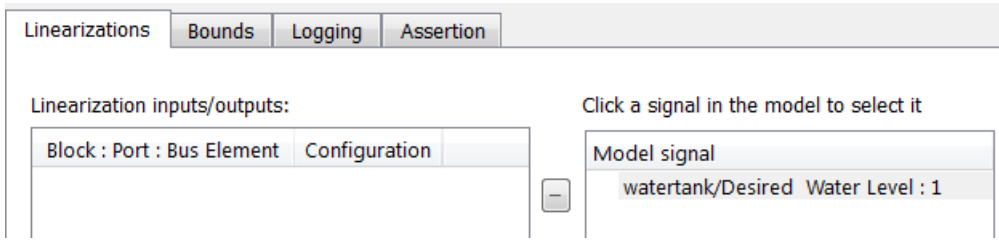
i Click adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.

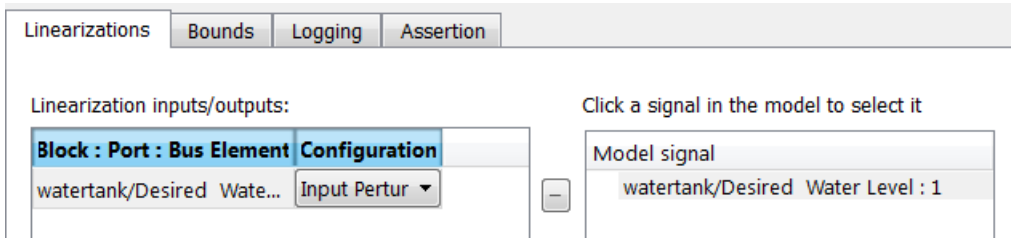


ii In the Simulink model, click the output signal of the `Desired Water Level` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



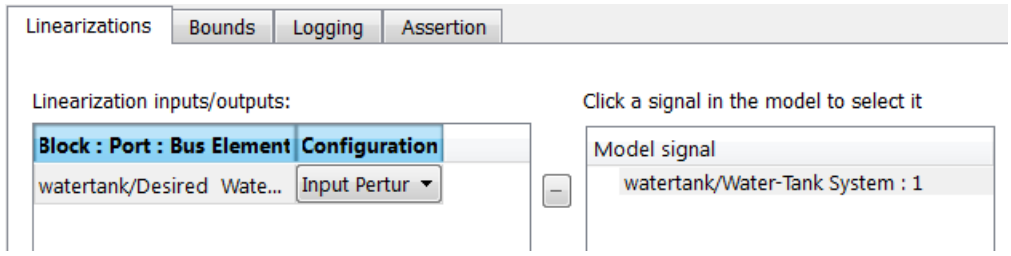
iii Click to add the signal to the **Linearization inputs/outputs** table.



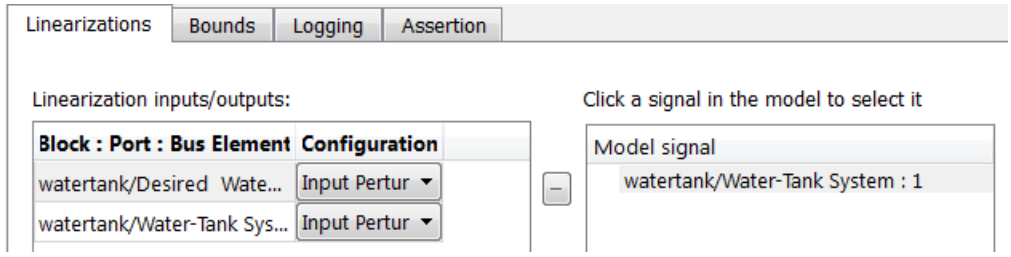
b To specify an output:

iv 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.

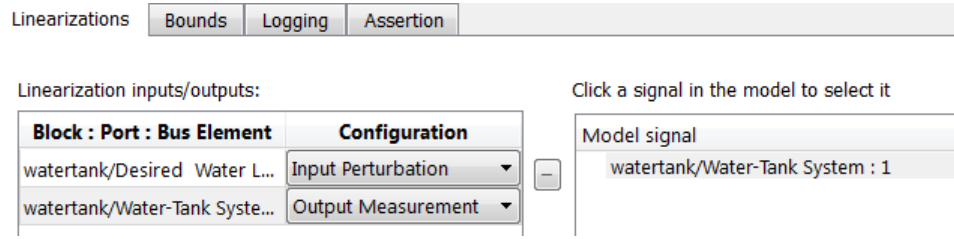


v Click to add the signal to the **Linearization inputs/outputs** table.



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 .

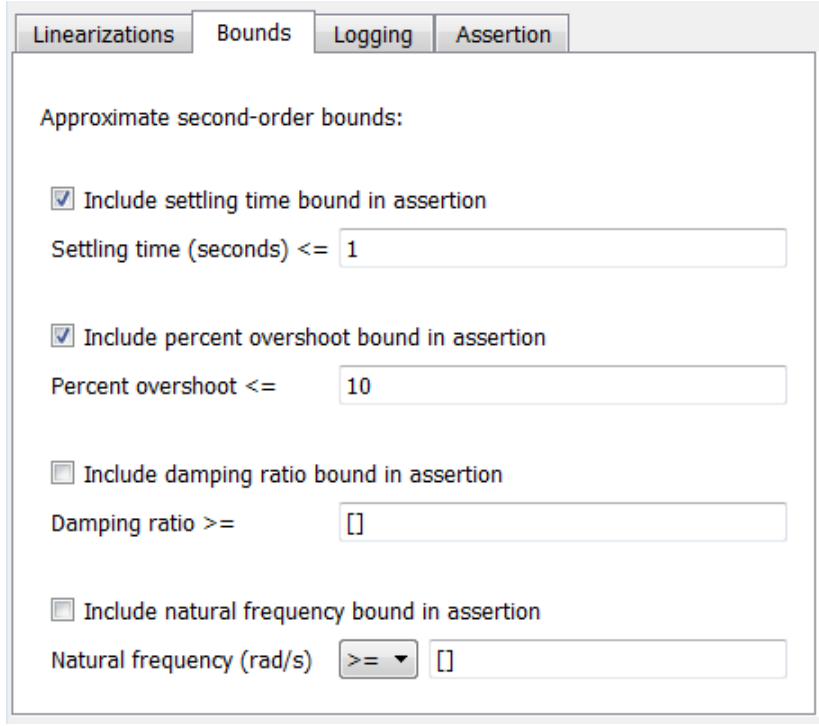
vi In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Output Measurement` for **watertank/Water-Tank System: 1**.



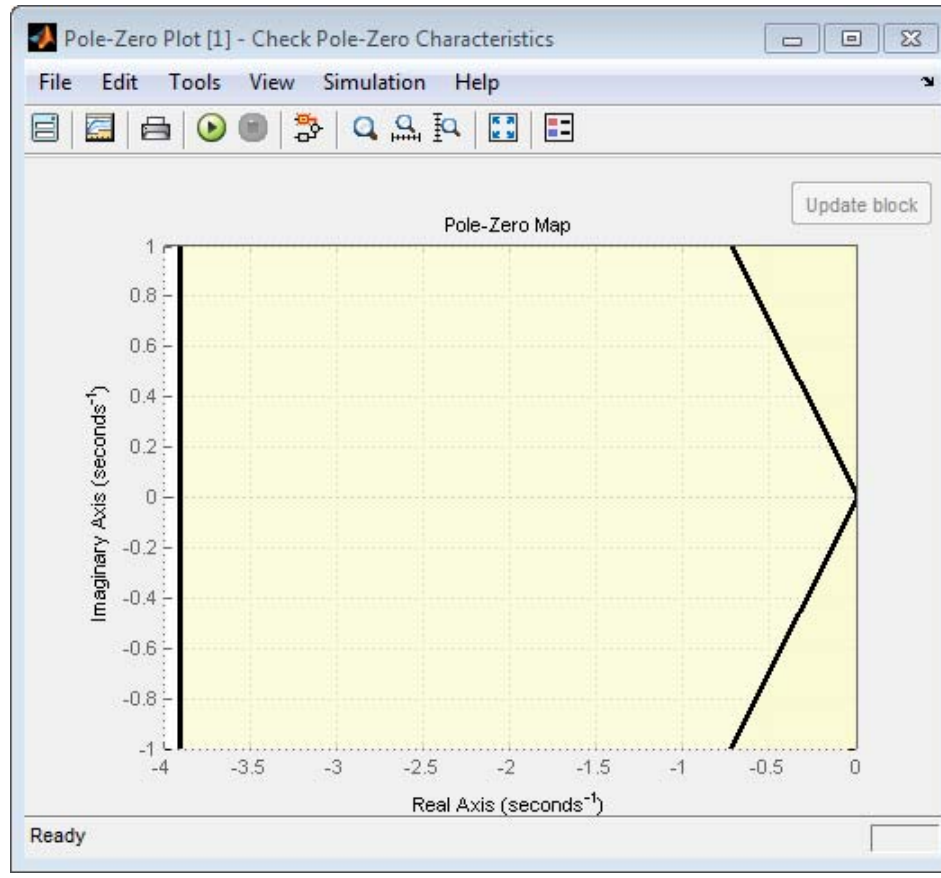
Note The I/Os include the feedback loop in the Simulink model. The software computes the poles and zeros of the closed-loop system.

vii Click to collapse the **Click a signal in the model to select it** area.

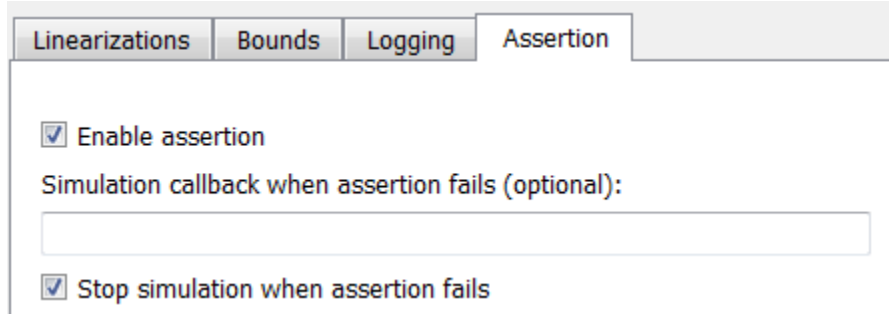
6 Specify bounds for assertion. In this example, you use the default approximate second-order bounds, specified in **Bounds** tab of the Block Parameters dialog box.



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.



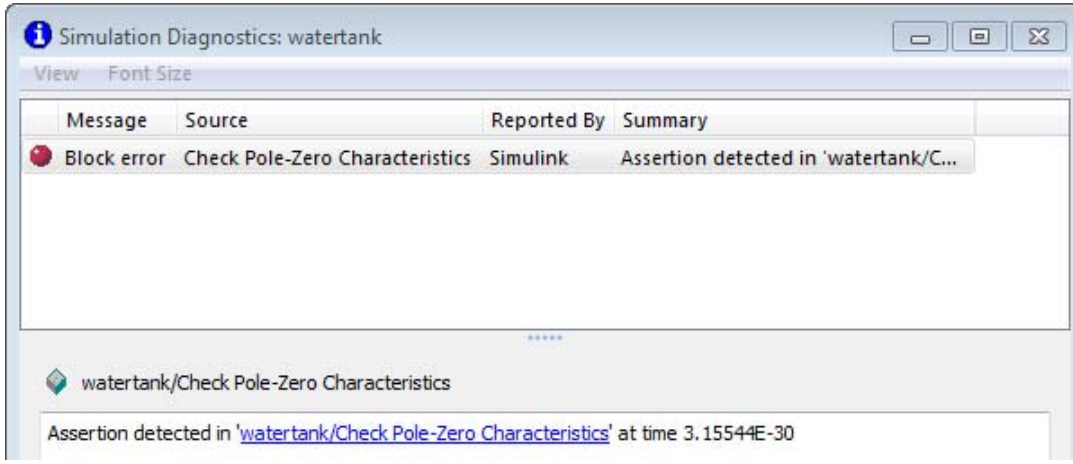
8 Click **Apply** to apply all changed settings to the block.

9 Simulate the model by clicking in the plot window.

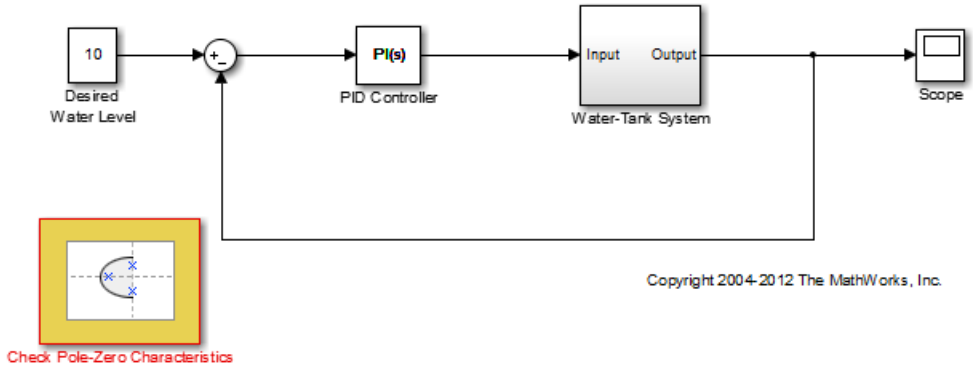
Alternatively, you can simulate the model from the Simulink Editor.

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 and the following windows appear:

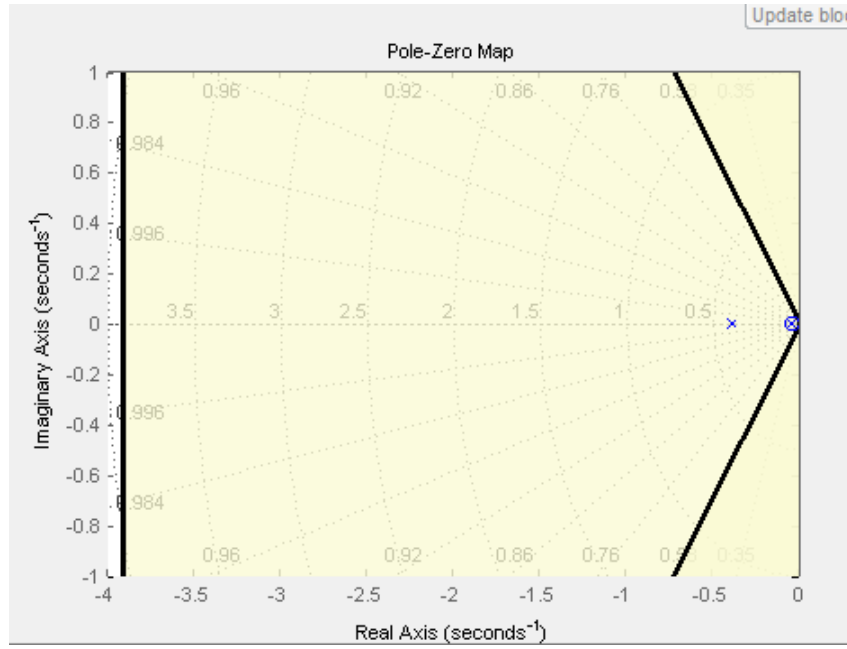
- Simulation Diagnostics window, reporting the block that asserts.



- Simulink Editor, highlighting the block that asserts.



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.



Model Verification 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

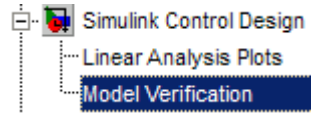
1 Open a nonlinear Simulink model. For example:

```
watertank
```

2 Open the Simulink Library Browser by selecting **View > Library Browser** in the Simulink Editor.

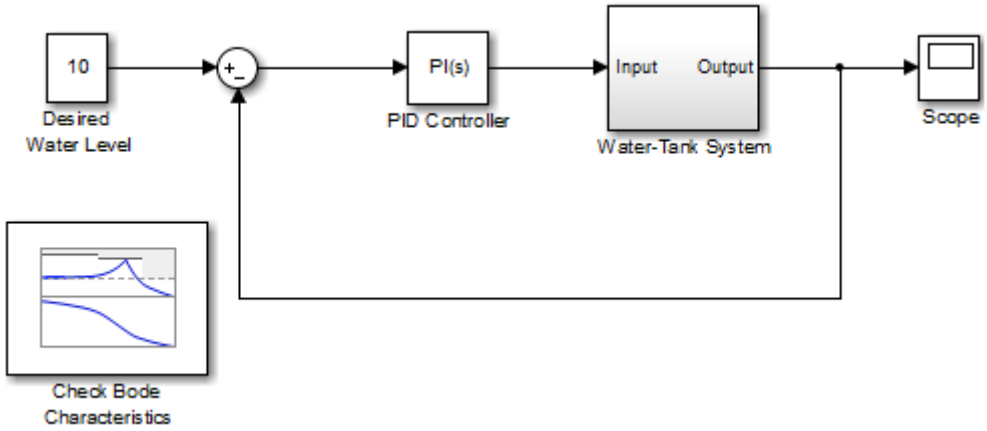
3 Add a model verification block to the Simulink model.

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.

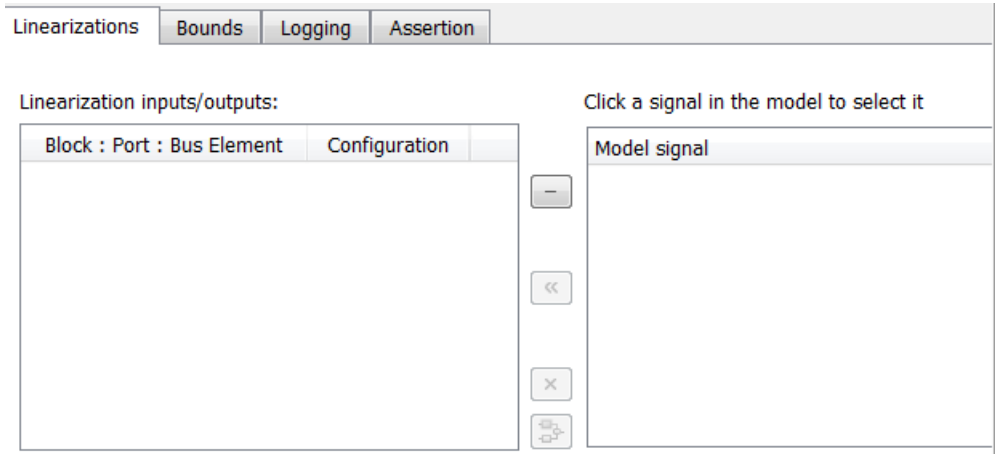
Tip If your model already contains I/O points, the block automatically detects these points and displays them.

---

a To specify an input:

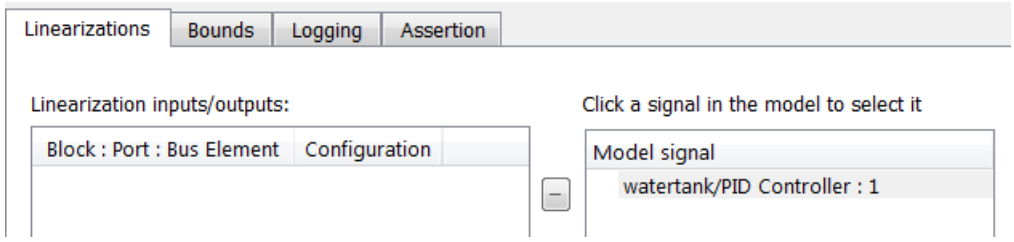
i Click adjacent to the **Linearization inputs/outputs** table.

The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.

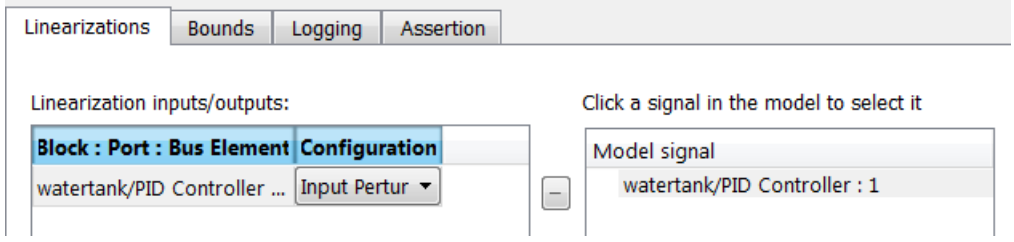


ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

The **Click a signal in the model to select it** area updates to display the selected signal.



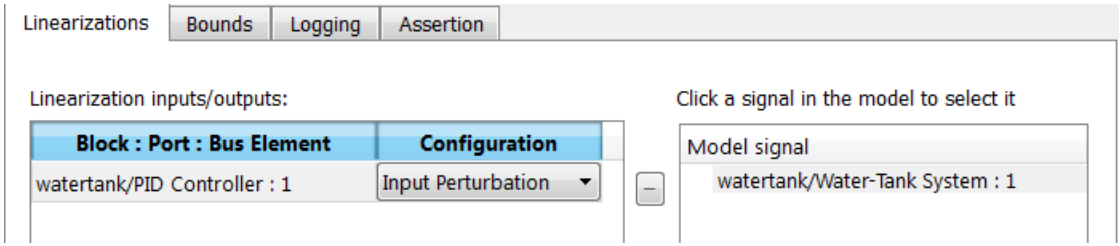
iii Click to add the signal to the **Linearization inputs/outputs** table.



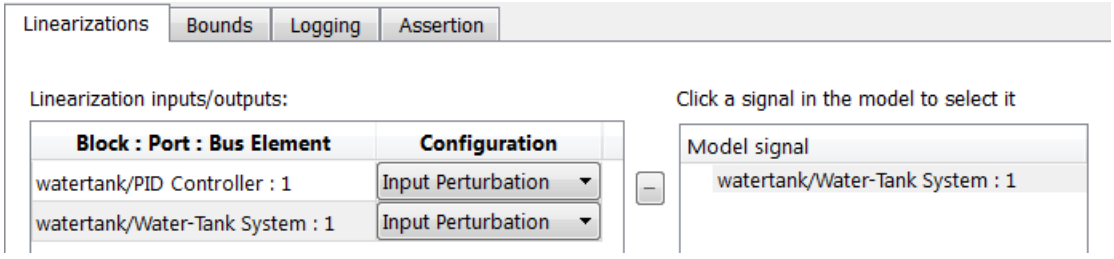
b To specify an output:

iv 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.



v Click to add the signal to the **Linearization inputs/outputs** table.

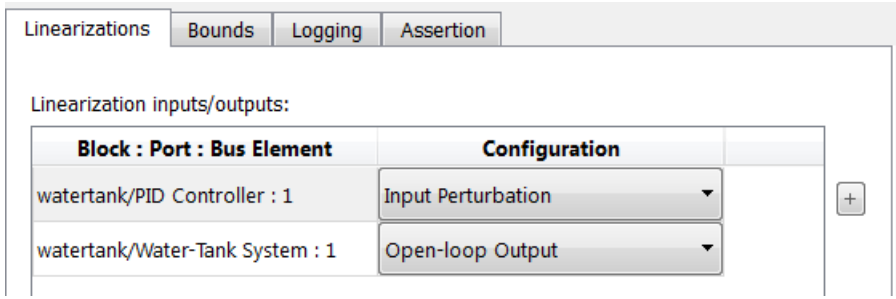


Note To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

---

vi In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `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.

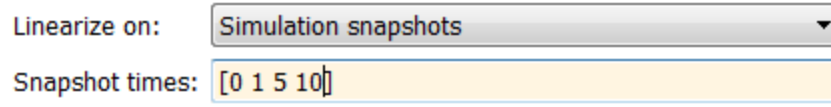
Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear in the **Linearization inputs/outputs** table.

---

6 Specify simulation snapshot times.

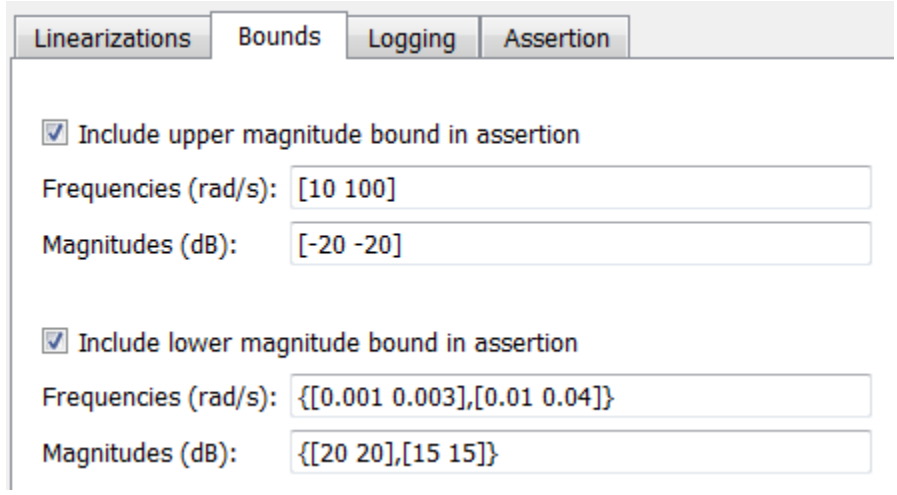
a In the **Linearizations** tab, verify that `Simulation snapshots` is selected in **Linearize on**.

b In the **Snapshot times** field, type `[0 1 5 10]`.



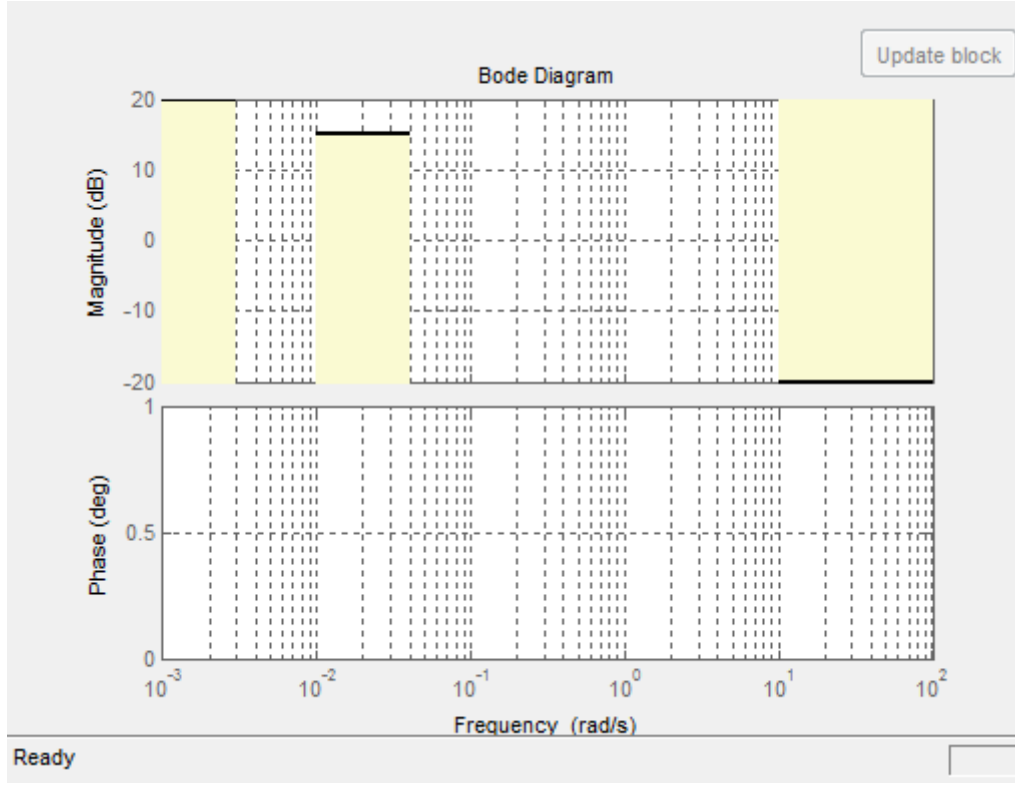
7 Specify multiple bound segments for assertion in the **Bounds** tab of the Block Parameters dialog box. In this example, enter the following lower magnitude bounds:

- **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.



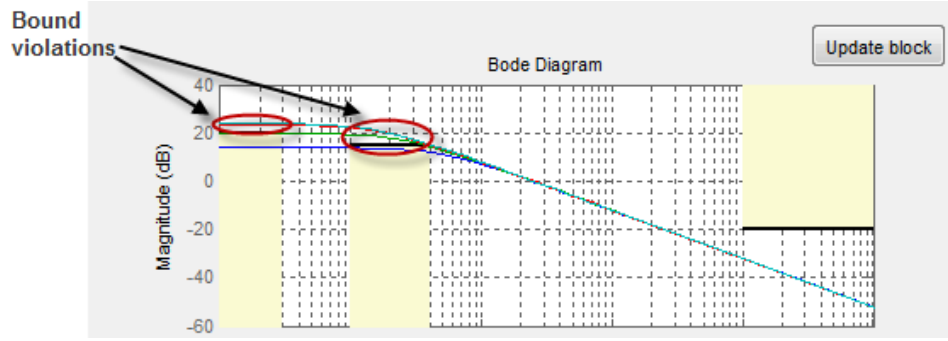
8 Simulate the model by clicking in the plot window.

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, the following warning messages appear at the MATLAB prompt:

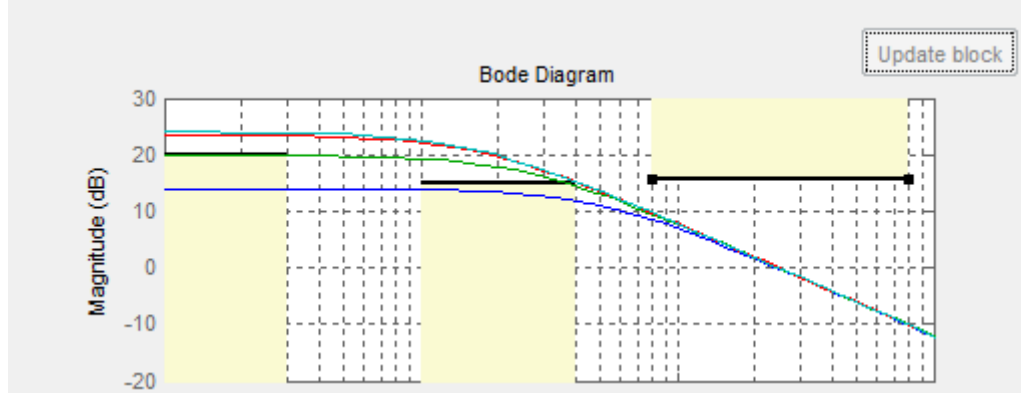
```
Warning: Assertion detected in 'watertank_check_bode/Check Bode
Characteristics' at time 0
Warning: Assertion detected in 'watertank_check_bode/Check Bode
Characteristics' at time 1
```

You can also view the bound violations in the plot window.

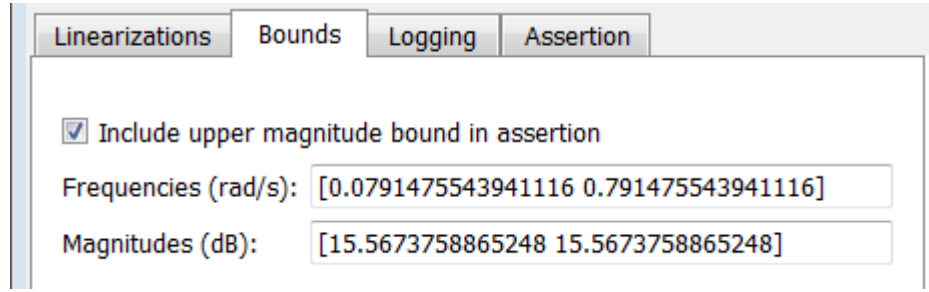


9 Modify a bound graphically. For example, to modify the upper magnitude bound graphically:

a In the plot window, click the bound segment to select it and then drag it to the desired location.

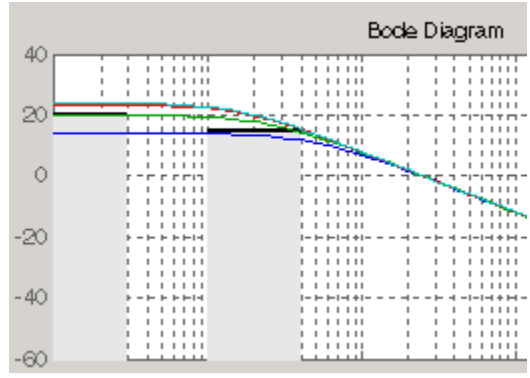


b Click **Update block** to update the new values in the Bounds tab of the Block Parameters dialog box.



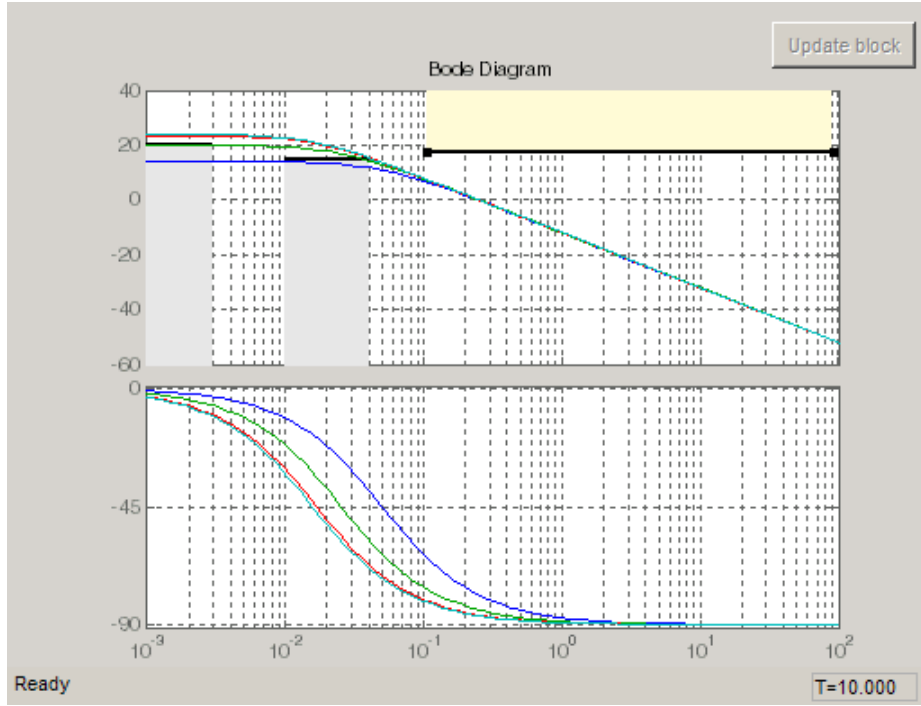
10 Disable the lower bounds to exclude them from asserting. Clear the **Include lower magnitude bounds in assertion** option in the Block Parameters dialog box. Then, click **Apply**.

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.


Update block
Bode Diagram
40
20
0
-20
-40
-60
0
-45
-90
10^-3
10^-2
10^-1
10^0
10^1
10^2
Ready
T=10.000

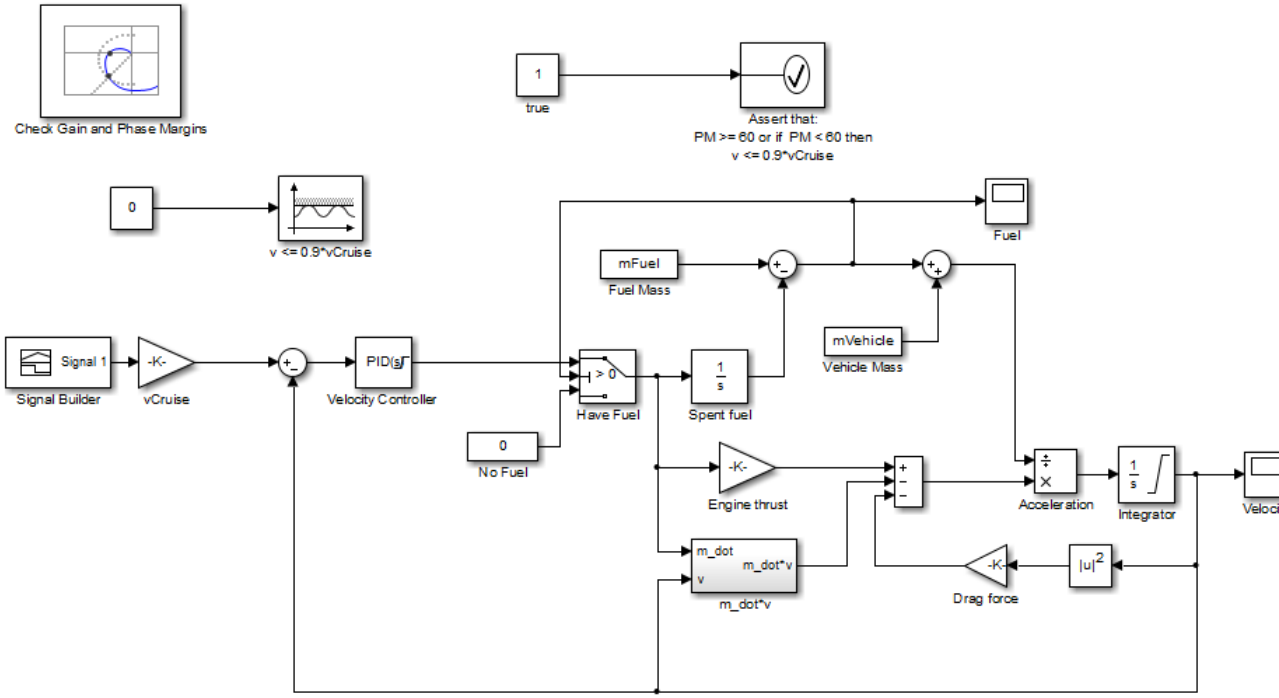
Model Verification Using Simulink Control Design and Simulink Verification Blocks

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.

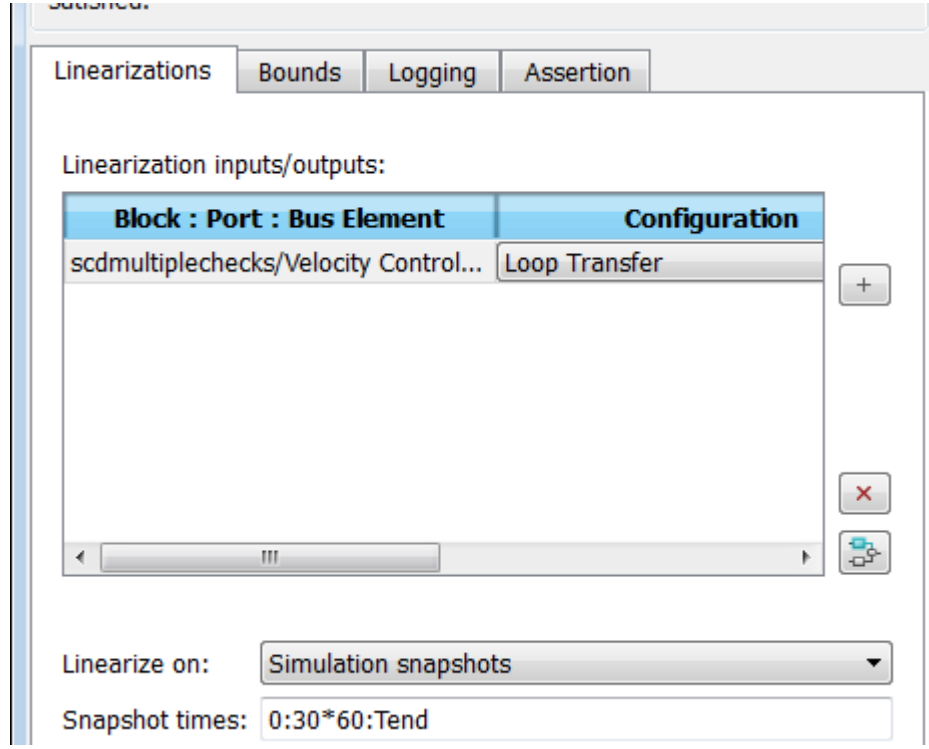
The model includes blocks to model:

- 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.

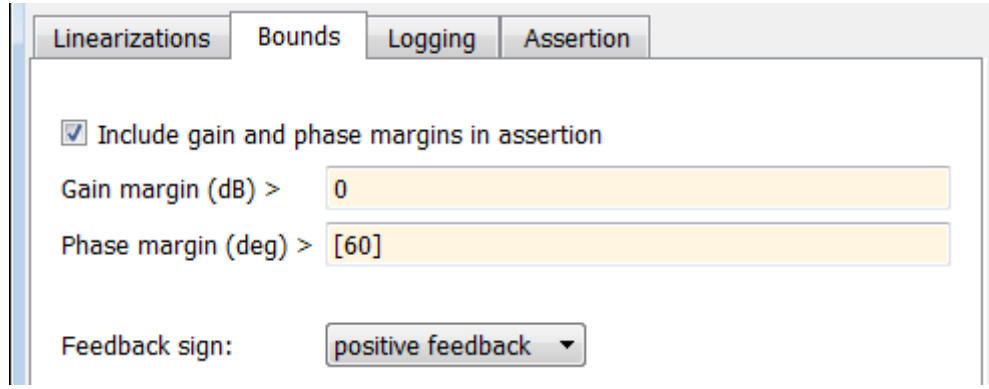


The model has already been configured with:

- 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`.

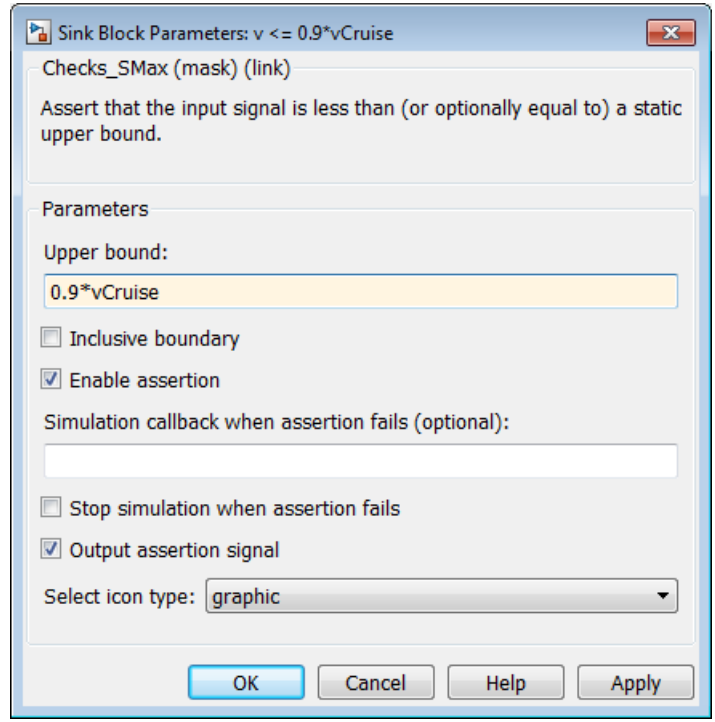
To view the phase margins to be computed later during simulation, specify `Tabular` in **Plot type**, and click **Show Plot**.

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, select **Output assertion signal** and click **Apply**.

      This action adds an output port z^{-1} to the block.

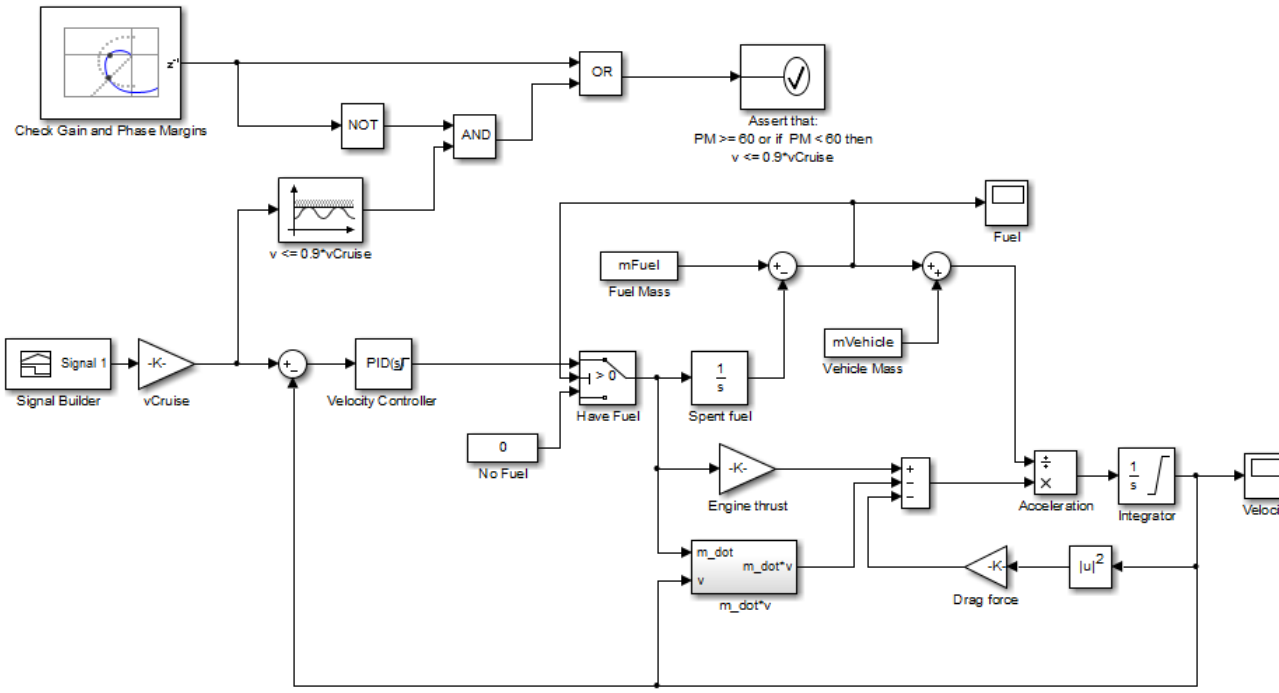
   b Double-click the `v <= 0.9*vCruise` block and specify the block parameters, as shown in the following figure. After setting the parameters, click **Apply**.



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 and warnings appear at the MATLAB prompt.
- The Check Gain and Phase Margins block asserts two times and warnings appear at the MATLAB prompt. You can also view the phase margins that violate the bound in the plot window.

- The `Assert that:  PM >= 60 or if PM < 60 then v <= 0.9*vCruise` does not encounter the assertion condition. Therefore, no warnings appear at the MATLAB prompt for this block and the simulation does not stop.

6

Class Reference

slTunable

Purpose

Interface for control system tuning of Simulink models

Description

`slTunable` provides an interface between a Simulink model and the tuning commands `slTunable.systune` and `slTunable.looptune`. (Using these tuning commands requires Robust Control Toolbox software). `slTunable` allows you to:

- Specify the control architecture.
- Extract information about the control system, such as linearized open-loop and closed-loop responses.
- Designate and parametrize blocks to be tuned.

Because tuning commands such as `systune` operate on linear models, the `slTunable` interface automatically computes and stores a linearization of your Simulink model. (For more information about linearization, see `linearize`). `slTunable` also parametrizes the tunable portion of your control system. You can use `slTunable` and its methods for tuning and analyzing the control system.

Commands that query the linearization stored in the `slTunable` interface automatically relinearize the Simulink model if you have changed any properties of the `slTunable` interface since the last linearization. Such commands include `slTunable.systune`, `slTunable.looptune`, `slTunable.getIOTransfer`, and `slTunable.getLoopTransfer`.

Construction

`ST = slTunable(mdl,tunedblocks)` creates an `slTunable` interface `ST` for tuning the blocks `tunedblocks` of the Simulink model `mdl`. Creating `ST` automatically parametrizes the blocks specified by `tunedblocks`. The interface `ST` also stores a linearization of `mdl`, computed at the operating point and linearization I/O points specified in the model.

`ST = slTunable(mdl,tunedblocks,io)` further specifies a vector `io` of linearization I/O points for analysis and tuning of the control system.

`ST = slTunable(mdl,tunedblocks,io,op)` also specifies an operating point `op` for linearizing the model.

`ST = slTunable(...,options)` specifies additional options for linearizing `mdl`.

`ST = slTunable(...,Name,Value)` sets the additional properties of `ST` specified by one or more `Name,Value` pair arguments. `Name` is a property name, and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Specifying properties of `ST` in this way saves repeated linearization by ensuring that the interface is fully configured before linearizing the model.

Input Arguments

mdl

String specifying name of Simulink model.

tunedblocks

String or cell array of strings specifying the Simulink block or blocks to tune. You can specify an abbreviated block name provided that the string matches the end of the full block path and unambiguously identifies the block to tune.

`slTunable` automatically parametrizes the blocks listed in `tunedblocks`. To override the default parametrization, use `slTunable.setBlockParam`.

io

Vector of I/O points for linearization, tuning, and analysis of the control system. (For more information about I/O points, see `linio`.) These I/O points specify which signals are model inputs, outputs, and loop opening locations. Only signals specified in `io` or added with `slTunable.addIO` are available in the interface `ST` for tuning and analysis.

op

Operating point specification for linearizing `mdl`. For more information about operating point specifications, see `operpoint` and `findop`.

options

Option set specifying additional options for the linearization of `mdl`. Use `linearizeOptions` to create `options`. For information about available options, see the `linearizeOptions` reference page.

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`.

Controls

Cell array of strings specifying the controller outputs that drive the actuators in the control system. These strings correspond to signal names in `mdl`.

The `Controls` and `Measurements` properties of `ST` define the boundary between the plant and controller in `mdl` for tuning with `slTunable.looptune`.

Use `slTunable.addControl` to specify additional control signals in the `Controls` property of an existing `slTunable` interface.

Default: 0-by-1 cell array

IOs

Cell array of strings specifying the input and output signals available for analysis and tuning. Each signal is characterized by:

- Full path in the model, either:

  ```
  BlockPath/PortNumber[SignalName]
  ```

  or

  ```
  BlockPath/PortNumber[BusElement1.BusElement2]
  ```

- Linearization I/O type `'input'`, `'output'`, `'sensitivity'`, or `'compsensitivity'`.

`ST.IOs` automatically includes all the linearization I/O points specified in the model. To remove a linearization I/O from `ST.IOs`, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.IOs`.

```
ST.IOs(2,:) = [];
```

To add a linearization I/O to `ST`, use `slTunable.addIO`.

For more information about linearization I/O points, see `linio`.

Default: 0-by-2 cell array

LinearizeOptions

Options set for linearizing `mdl`. Use `linearizeOptions` to create the options set. Use `slTunable.linearize` to relinearize `mdl` and update the `slTunable` interface after changing `LinearizeOptions`.

Measurements

Cell array of strings specifying the sensor outputs that feed the controller in the control system. These strings correspond to signal names in `mdl`. The `Controls` and `Measurements` properties of `ST` define the boundary between the plant and controller in `mdl` for tuning with `slTunable.looptune`.

Use `slTunable.addMeasurement` to specify additional measurement signals in the `Measurement` property of an existing `slTunable` interface.

Default: 0-by-1 cell array

ModelName

String specifying the name of the model being analyzed or tuned. `ModelName` is set by the input argument `mdl`.

Notes

User notes. This property can store any text as a string or cell array of strings. For example:

```
ST.Notes = 'This model was created on Jan. 1st, 2011.';
```

Default: `{}`

Openings

Cell array of strings specifying signal locations in `mdl` where feedback loops are broken. The loop openings are enforced in all tuning and validation commands such as `slTunable.looptune` and `slTunable.loopview`.

Loop opening locations are stored in the `Openings` property as the full path name of the corresponding signal in `mdl`, either:

```
BlockPath/PortNumber[SignalName]
```

or

```
BlockPath/PortNumber[BusElement1.BusElement2]
```

Use `slTunable.addOpening` to add additional loop openings to the `Opening` property of an existing `slTunable` interface. To remove a loop opening location, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.Openings`.

```
ST.Openings(2,:) = [];
```

Default: 0-by-1 cell array

OperatingPoint

Operating point specification for linearizing `mdl`. Use `slTunable.linearize` to relinearize `mdl` and update the `slTunable` interface after changing `OperatingPoint`.

For more information about operating point specifications, see `operpoint` and `findop`.

Ts

Working sampling time for analysis and tuning. `mdl` is linearized at this sampling time.

Default: 0

Switches

Cell array specifying signal locations in `mdl` where feedback loops may be broken for analysis or tuning purposes. `Switches` differs from `Openings` in that the loop openings specified in `Openings` are always enforced in tuning and analysis. In contrast, locations specified in `Switches` are closed by default unless you explicitly specify them open. You can specify switch locations as openings in commands such as `slTunable.getIOTransfer`, `slTunable.getLoopTransfer`, or `TuningGoal.LoopShape`.

Switch locations are stored in the `Switches` property as the full path name of the corresponding signal in `mdl`, either:

```
BlockPath/PortNumber[SignalName]
```

or

```
BlockPath/PortNumber[BusElement1.BusElement2]
```

Use `slTunable.addSwitch` to add additional loop openings to the `Switches` property of an existing `slTunable` interface. To remove a loop opening location, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.Switches`.

```
ST.Switches(2,:) = [];
```

Default: 0-by-1 cell array

TunedBlocks

Cell array of strings specifying full paths to the tuned blocks in `mdl`.

When you create an `slTunable` interface, the `TunedBlocks` property is automatically populated with the blocks you specify in the `tunedblocks` input. To specify additional tunable blocks in an existing an existing `slTunable` interface, use `slTunable.addBlock`

Properties

Controls

Cell array of strings specifying the controller outputs that drive the actuators in the control system. These strings correspond to signal names in `mdl`.

The `Controls` and `Measurements` properties of `ST` define the boundary between the plant and controller in `mdl` for tuning with `slTunable.looptune`.

Use `slTunable.addControl` to specify additional control signals in the `Controls` property of an existing `slTunable` interface.

Default: 0-by-1 cell array

IOs

Cell array of strings specifying the input and output signals available for analysis and tuning. Each signal is characterized by:

- Full path in the model, either:

  ```
  BlockPath/PortNumber[SignalName]
  ```

  or

  ```
  BlockPath/PortNumber[BusElement1.BusElement2]
  ```

- Linearization I/O type `'input'`, `'output'`, `'sensitivity'`, or `'compsensitivity'`.

`ST.IOs` automatically includes all the linearization I/O points specified in the model. To remove a linearization I/O from `ST.IOs`, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.IOs`.

```
ST.IOs(2,:) = [];
```

To add a linearization I/O to `ST`, use `slTunable.addIO`.

For more information about linearization I/O points, see `linio`.

Default: 0-by-2 cell array

LinearizeOptions

Options set for linearizing `mdl`. Use `linearizeOptions` to create the options set. Use `slTunable.linearize` to relinearize `mdl` and update the `slTunable` interface after changing `LinearizeOptions`.

Measurements

Cell array of strings specifying the sensor outputs that feed the controller in the control system. These strings correspond to signal names in `mdl`. The `Controls` and `Measurements` properties of `ST` define the boundary between the plant and controller in `mdl` for tuning with `slTunable.looptune`.

Use `slTunable.addMeasurement` to specify additional measurement signals in the `Measurement` property of an existing `slTunable` interface.

Default: 0-by-1 cell array

ModelName

String specifying the name of the model being analyzed or tuned. `ModelName` is set by the input argument `mdl`.

Notes

User notes. This property can store any text as a string or cell array of strings. For example:

```
ST.Notes = 'This model was created on Jan. 1st, 2011.';
```

Default: `{}`

Openings

Cell array of strings specifying signal locations in `mdl` where feedback loops are broken. The loop openings are enforced in all tuning and validation commands such as `slTunable.looptune` and `slTunable.loopview`.

Loop opening locations are stored in the `Openings` property as the full path name of the corresponding signal in `mdl`, either:

```
BlockPath/PortNumber[SignalName]
```

or

```
BlockPath/PortNumber[BusElement1.BusElement2]
```

Use `slTunable.addOpening` to add additional loop openings to the `Opening` property of an existing `slTunable` interface. To remove a loop opening location, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.Openings`.

```
ST.Openings(2,:) = [];
```

Default: 0-by-1 cell array

OperatingPoint

Operating point specification for linearizing `mdl`. Use `slTunable.linearize` to relinearize `mdl` and update the `slTunable` interface after changing `OperatingPoint`.

For more information about operating point specifications, see `operpoint` and `findop`.

Ts

Working sampling time for analysis and tuning. `mdl` is linearized at this sampling time.

Default: 0

Switches

Cell array specifying signal locations in `mdl` where feedback loops may be broken for analysis or tuning purposes. `Switches` differs from `Openings` in that the loop openings specified in `Openings` are always enforced in tuning and analysis. In contrast, locations specified in `Switches` are closed by default unless you explicitly specify them open. You can specify switch locations as openings in commands such as `slTunable.getIOTransfer`, `slTunable.getLoopTransfer`, or `TuningGoal.LoopShape`.

Switch locations are stored in the `Switches` property as the full path name of the corresponding signal in `mdl`, either:

```
BlockPath/PortNumber[SignalName]
```

or

```
BlockPath/PortNumber[BusElement1.BusElement2]
```

Use `slTunable.addSwitch` to add additional loop openings to the `Switches` property of an existing `slTunable` interface. To remove a loop opening location, delete a row from the cell array. For example, the following command deletes the second I/O from `ST.Switches`.

```
ST.Switches(2,:) = [];
```

Default: 0-by-1 cell array

TunedBlocks

Cell array of strings specifying full paths to the tuned blocks in `mdl`.

When you create an `slTunable` interface, the `TunedBlocks` property is automatically populated with the blocks you specify in the `tunedblocks` input. To specify additional tunable blocks in an existing an existing `slTunable` interface, use `slTunable.addBlock`

Methods

addBlock	Add block to list of tuned blocks
addControl	Add control signal to list of control signals in interface
addIO	Add I/O points list of linearization I/Os in interface
addMeasurement	Add measurement signal to list of measurement signals in interface
addOpening	Add new opening location to list of loop openings in interface
addSwitch	Add switch location to list of loop opening switches in interface

getBlockParam	Get parametrization of a particular tuned block in `slTunable` interface
getBlockRateConversion	Query rate conversion method of tunable block
getBlockValue	Current value of a particular tuned block in `slTunable` interface
getIOTransfer	Tunable model of closed-loop transfer function between specified inputs and outputs
getLoopTransfer	Tunable model of open-loop transfer function at specified location
linearize	Relinearize Simulink model to update `slTunable` interface
looptune	Tune MIMO feedback loops in Simulink
looptuneSetup	Convert tuning setup for `looptune` to tuning setup for `systune`
loopview	Graphically analyze results of control system tuning
readBlockValue	Update tuned block values from Simulink model
setBlockParam	Specify parametrization for specified tuned block
setBlockRateConversion	Set rate conversion method of tunable block

setBlockValue	Set current value of parametrization for specified block
showBlockValue	Display current value of block parametrizations
showTunable	Display current value of block parametrizations
systune	Tune fixed-structure control system parameters in Simulink
writeBlockValue	Update block values in Simulink model

Copy Semantics

To make an independently modifiable copy of an `slTunable` interface, use the `copy` command. For example:

```
ST = copy(STO)
```

creates an `slTunable` interface `ST` that initially contains identical values to `STO` for all properties. You can now modify `ST` without changing `STO`.

`slTunable` is a handle class. To learn more about how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

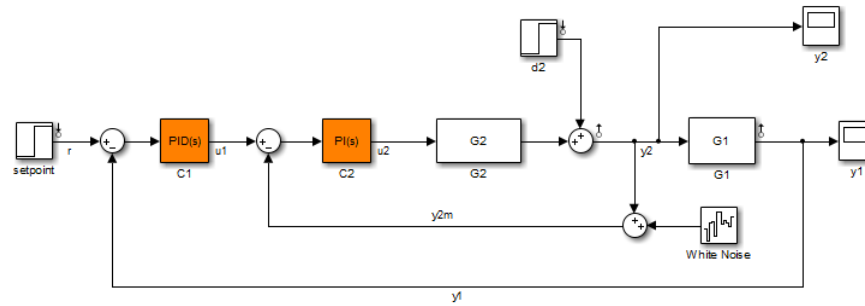
Examples

Create and Configure slTunable Interface

Create and configure a `slTunable` interface for tuning the Simulink model `rct_cascade` using `slTunable.looptune`.

Open the Simulink model.

```
open_system('rct_cascade')
```



Create the `slTunable` interface for `rct_cascade`.

```
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);
```

The tunable blocks of this model are the PID controllers, `C1` and `C2`. This command creates the `slTunable` interface `ST0` with those tunable blocks.

Specify the control and measurement signals of the control system.

```
addControl(ST0,'u2');
addMeasurement(ST0,{'y1','y2m'});
```

The commands `slTunable.addControl` and `slTunable.addMeasurement` add the specified signals to the `Controls` and `Measurements` properties of `ST0`.

Control and measurement signals define the boundary between plant and controller for tuning with `slTunable.looptune`. In `rct_cascade`, the actuator signal is `u2`, the output of the inner-loop controller. The measurement signals are `y1` and `y2m`, each of which feeds into a portion of the cascaded controller.

Specify the signal `y1` as a location for an optional loop opening (a switch).

```
addSwitch(ST0,'y1');
```

When tuning a cascade control system, it can be useful to impose certain requirements on the inner loop that are enforced with the outer loop open. However, such a control system also has overall requirements are enforced with all loops closed. Therefore, add a switch location, instead of a loop opening location.

You can now use `STO` to tune the control system with `slTunable.looptune`.

Algorithms

`slTunable` linearizes your Simulink model using the algorithms described in "Exact Linearization Algorithm" on page 2-152.

Tutorials

- "Tuning Control Systems in Simulink"

How To

- "Set Up Your Control System for Tuning with systune"
- "Specifying Design Requirements for systune"
- "Set Up Your Control System for Tuning with looptune"
- "Performance and Robustness Specifications for looptune"

7

Alphabetical List

addoutputspec

Purpose

Add output specification to operating point specification

Syntax

```
opnew=addoutputspec(op,'block',portnumber)
```

Alternatives

As an alternative to the `addoutputspec` function, add output specifications with the Simulink Control Design GUI. See "Steady-State Operating Point to Meet Output Specification" on page 1-21.

Description

`opnew=addoutputspec(op,'block',portnumber)` adds an output specification for a Simulink model to an existing operating point specification, `op`, created with `operspec`. The signal being constrained by the output specification is indicated by the name of the block, `'block'`, and the port number, `portnumber`, that it originates from.

You can edit the output specification within the new operating point specification object, `opnew`, to include the actual constraints or specifications for the signal. Use the new operating point specification object with the function `findop` to find operating points for the model.

This function automatically compiles the Simulink model, given in the property `Model` of `op`, to find the block's output portwidth.

Examples

Create an operating point specification for the model `magball`.

```
op=operspec('magball')
```

This specification returns the object `op`. Note that there are no outports in this model and no outputs in the object `op`.

```
 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:             0
(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:             0
(5.) magball/Magnetic Ball Plant/height
      spec:  dx = 0,  initial guess:          0.05

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

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

To add an output specification to the signal between the Controller block and the Magnetic Ball Plant block, use the function `addoutputspec`.

```
newop=addoutputspec(op,'magball/Controller',1)
```

This function adds the output specification is added to the operating point specification object.

```
  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:             0
(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:             0
(5.) magball/Magnetic Ball Plant/height
      spec:  dx = 0,  initial guess:          0.05
```

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

Outputs:
----------
(1.) magball/Controller
      spec:  none
```

Edit the output specification to constrain this signal to be 14.

```
newop.Outputs(1).Known=1, newop.Outputs(1).y=14
```

The final output specification 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:             0
(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:             0
(5.) magball/Magnetic Ball Plant/height
      spec:  dx = 0,  initial guess:          0.05

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

Outputs:
----------
(1.) magball/Controller
      spec:  y = 0
```

See Also

`findop` | `operspec` | `operpoint`

Purpose

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`

findop

Purpose

Steady-state operating point from specifications (trimming) or simulation

Syntax

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

Description

`[op,opreport] = findop(sys,opspec)` returns the steady-state operating point of the model that meets the specifications `opspec`. The Simulink model must be open. If `opspec` is a vector of operating points specifications, `findop` returns a vector of corresponding operating points.

`[op,opreport] = findop(sys,opspec,options)` searches for the operating point of the model using additional optimization algorithm options specified by `options`.

`op = findop(sys,tsnapshot)` simulates the model and extracts operating points at the simulation snapshot time instants (snapshots) `tsnapshot`.

Tips

- Initialize 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`.

Input Arguments

sys

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

opspec

Operating point specification object for the model `sys`, specified using `operspec`.

`opspec` can also be a vector of operating point specification objects. If `opspec` is a vector of operating points specifications, `findop` returns a vector of corresponding operating points. In this case, `findop` needs to compile the model only once. Using a vector of operating points allows

you to find multiple trimmed operating points without the overhead of compiling the model for each trimming computation.

options

Algorithm options, specified using `findopOptions`.

tsnapshot

Simulation snapshot time instants when to extract the operating point of the model, specified as a scalar or vector.

Output Arguments

op

Operating point object.

After creating the operating point object, you can modify the operating point states and input levels. For example, `op.States(1).x` stores the state values of the first model state, and `op.Inputs(1).u` stores the input level of the first inport block.

The operating point object has these properties:

- `Model` — Simulink model name. String.
- `States` — State operating points of the Simulink model. Vector of data structures, where each data structure represents the states of one Simulink block. Each `States` structure has these fields:

`x`	Simulink block state values, specified as a vector of states.
`Ts`	(Only for discrete-time states) Sample time and offset of each Simulink block state, specified as a vector.
`Description`	Block state description, specified as a string.
`Nx` (read only)	Number of states in the Simulink block.
`Block`	Simulink block name.

`SampleType`	State time rate can have the values: • `'CSTATE'` — Continuous-time state • `'DSTATE'` — Discrete-time state.
`inReferencedModel`	Determine whether the sates is inside a reference model: • `1` — State is inside a reference model. • `0` — State is in the current model file.

- `Inputs` — Input levels at the operating point. Vector of data structures, where each data structure represents the input levels of one root-level inport block in the Simulink block. Each `Inputs` data structure has these fields:

`u`	Inport block input levels at the operating point, specified as a vector of input levels.
`Description`	Inport block input description, specified as a string.
`Block`	Inport block name.
`PortWidth`	Number of inport block signals.

- `Time` — Time instants for evaluating the time-varying functions in the model.

If `opspec` is a vector of operating point specification objects, then `op` is a vector of corresponding operating points.

opreport

Optimization results report object.

This report displays automatically even when you suppress the output using a semicolon. You can avoid displaying the report by using `findopOptions` to set the `DisplayReport` field in `options` to `'off'`.

The `opreport` object has these properties:

- `Model` — `Model` property value of the `op` object.
- `Inputs` — `Inputs` property value of the `op` object.
- `Outputs` — `Outputs` property value of the `op` object with the addition of `yspec`, which is the desired `y` value.
- `States` — `States` property value of the `op` object with the addition of `dx`, which are the state derivative values.
- `Time` — `Time` property value of the `op` object.
- `TerminationString` — Optimization termination condition, stored as a string.
- `OptimizationOutput` — Optimization algorithm results, returned as a structure with these fields:

`iterations`	Number of iterations performed during the optimization
`funcCount`	Number of function evaluations performed during the optimization
`lssteplength`	Size of line search step relative to search direction (active-set optimization algorithm only)
`stepsize`	Displacement in the state vector at the final iteration (active-set and interior-point optimization algorithms)
`algorithm`	Optimization algorithm used

`firstorderopt`	Measure of first-order optimality, for the trust-region-reflective optimization algorithm; `[ ]` for other algorithms
`constrviolation`	Maximum of constraint functions
`message`	Exit message

For more information about the optimization algorithm, see the Optimization Toolbox™ documentation.

Definitions

Steady-State Operating Point (Trim Condition)

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

A model might have several steady-state operating points. For example, a hanging pendulum has two steady-state operating points. A *stable steady-state operating point* occurs when a pendulum hangs straight down. That is, the pendulum position does not change with time. When the pendulum position deviates slightly, the pendulum always returns to equilibrium; 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 a nonlinear system, your initial guesses for the states and input levels must be in the neighborhood of 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.

Examples

Steady-State Operating Point (Trimming) From Specifications

This example shows how to use `findop` to compute an operating point of a model from specifications.

1 Open Simulink model.

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

2 Create operating point specification object.

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

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

```
 Operating 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
----------
```

`operspec` extracts the default operating point of the Simulink model with two states. The model does not have any root-level inport blocks and no root-level outport blocks or output constraints.

3 Configure specifications for the first model state.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

The first state must be at steady state and have an initial value of 2 with a lower bound of 0.

4 Configure specifications for the second model state.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

The second state sets the desired height of the water in the tank at 10. Configuring the height as a known value keeps this value fixed when computing the operating point.

5 Find the operating point that meets these specifications.

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

`opreport` describes how closely the optimization algorithm met the specifications at the end of the operating point search.

```
 Operating 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` indicates the time derivative of each state. The actual `dx` values of zero indicate that the operating point is at steady state. The desired `dx` value is in parentheses.

Steady-State Operating Point to Meet Output Specification

This example shows how to specify an output constraint for computing the steady-state operating point of a model.

1 Open Simulink model.

```
sys = 'scdspeed';
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 Configure the output specification.

```
blk = [sys '/rad//s to rpm'];
opspec = addoutputspec(opspec,blk,1);
opspec.Outputs(1).Known = true;
opspec.Outputs(1).y = 2000;
```

`addoutputspec` adds to the operating point specification an output specification for the output of the block `rad/s to rpm`. This output specification, stored in `opspec.Outputs(1)`, allows you to specify a fixed output value for that block as part of the operating point specification. Setting the `Known` attribute of the output specification

to `true` ensures that the fixed output level is a constraint in the operating point search.

4 Find the operating point that meets the output specification.

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

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

 Operating 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 search report shows that the operating point search was successful. `op` is an operating point object that specifies a steady-state operating point for the model `scdspeed`, in which the output of the `rad/s to rpm` block is 2000.

Operating Points for Multiple Specification Sets

Find operating points for multiple operating point specifications with a single model compilation.

Each time you call `findop`, the software compiles the Simulink model. To find operating points for multiple specifications, you can give `findop` a vector of operating point specifications. Then `findop` only compiles the model once.

1 Open Simulink model.

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

2 Create operating point specification object.

```
opspec1 = operspec(sys);
```

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

3 Configure the output specification.

```
blk = [sys '/rad//s to rpm'];
opspec1 = addoutputspec(opspec1,blk,1);
opspec1.Outputs(1).Known = true;
opspec1.Outputs(1).y = 1500;
```

`opspec1` specifies a stead-state operating point in which the output of the block `rad/s to rpm` is fixed at 500.

Note Alternatively, you can configure an operating point specification using the Linear Analysis Tool and export the specification to the MATLAB workspace. See "Import and Export Specifications For Operating Point Search" on page 1-36 for more information.

---

4 Create and configure additional operating point specifications.

```
opspec2 = copy(opspec1);
opspec2.Outputs(1).y = 2000;

opspec3 = copy(opspec1);
opspec3.Outputs(1).y = 2500;
```

Using the `copy` command creates an independent operating point specification that you can edit without changing `opspec1`. Here, the specifications `opspec2` and `opspec3` are identical to `opspec1`, except for the target output level.

5 Find the operating points that meet each of the three output specifications.

```
opspecs = [opspec1,opspec2,opspec3];
ops = findop(sys,opspecs);
bdclose(sys);
```

Pass the three operating point specifications to `findop` in the vector `opspecs`. When you give `findop` a vector of operating point specifications, it finds all the operating points with only one model compilation. `ops` is a vector of operating point objects for the model `scdspeed` that correspond to the three specifications in the vector.

Initialize Steady-State Operating Point Search Using Simulation

This example shows how to use `findop` to compute an operating point of a model from specifications, where the initial state values are extracted from a simulation snapshot.

1 Open Simulink model.

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

2 Extract an operating point from simulation after 10 time units.

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

3 Create operating point specification object.

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

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

4 Configure initial values for operating point search.

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

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

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

`opreport` describes the optimization algorithm status at the end of the operating point search.

```
 Operating 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:     -1.1e-014 (0)

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

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

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

Steady-State Operating Points at Simulation Snapshots

This example shows how to use `findop` to extract operating points of a model from specifications snapshots.

1 Open Simulink model.

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

2 Extract an operating point from simulation at 10 and 20 time units.

```
op = findop(sys,[10,20]);
```

3 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-007
(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-008
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
```

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

4 Display the second operating point.

```
op(2)

 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=20)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 2.07e-007
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 3.19e-008
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

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

View Operating Point Object

This example shows how to use `get` to display the operating point states, inputs, and outputs.

```
sys = 'watertank';
load_system(sys);
op = operpoint(sys)
get(op.States(1))
```

Synchronize Simulink Model Changes With Operating Point Specification

This example shows how to use `update` to update an existing operating point specification object after you update the Simulink model.

1. Open Simulink model.

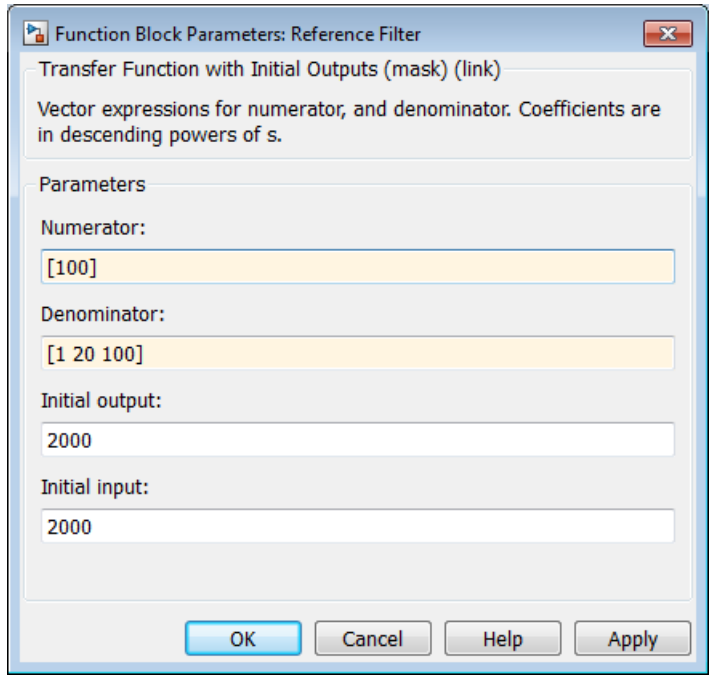
   ```
   sys = 'scdspeedctrl';
   open_system(sys);
   ```

2. Create operating point specification object.

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

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

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 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 poin
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);
```

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

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'`

After the optimization completes, Simulink restores the original model properties.

Alternatives

As an alternative to the `findop` command, find operating points using the Linear Analysis Tool. See the following examples:

- "Steady-State Operating Points from State Specifications" on page 1-16
- "Steady-State Operating Point to Meet Output Specification" on page 1-21

See Also

`initopspec` | `linearize` | `findopOptions` | `operspec` | `addoutputspec`

Tutorials

- "Steady-State Operating Point (Trimming)" on page 1-2
- "Choosing Between Simulation Snapshot and Operating Point from Specifications" on page 1-12
- "Simulink Model States Included in Operating Point Object" on page 1-4

How To

- "Steady-State Operating Points (Trimming) from Specifications" on page 1-14
- "Steady-State Operating Points from Simulation" on page 1-40

findopOptions

Purpose

Set options for finding operating points from specifications

Syntax

```
options = findopOptions
options = findopOptions(Name,Value)
```

Alternatives

As an alternative to `findopOptions` function, set options for finding operating points in the Linear Analysis Tool.

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.

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`.

`findopOptions` takes the following `Name` arguments:

'OptimizerType'

Optimizer type used by the optimization algorithm of `findop`, specified as one of the following strings:

- `'graddescent_elim'` — Enforces 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' value. The optimizer fixes the states, `x`, and inputs, `u`, that are marked as 'Known' in an operating point specification and then optimizes the remaining variables.

- `'graddescent'` — Enforces 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' value. `findop` also minimizes the error between the states, `x`, and inputs, `u`, that are marked as 'Known' in 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.
- `'lsqnonlin'` — Fixes the states, `x`, and inputs, `u`, that are marked as 'Known' in an operating point specification and optimizes the remaining variables. The algorithm then 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' value.
- `'simplex'` — Uses the same cost function as `lsqnonlin` with the direct search optimization routine found in `fminsearch`.

For more information about these optimization algorithms, see `fmincon` and `fminsearch` in the Optimization Toolbox documentation.

Default: `'graddescent_elim'`

'OptimizationOptions'

Options for the optimization algorithm, specified as a structure. Create the structure using the `optimset` command. For more information on these options, see the `optimset` reference page.

'DisplayReport'

Flag indicating whether to display the operating point summary report, specified as either `'off'` or `'on'`.

- `'on'` — Display the operating point summary report in the MATLAB command window when running `findop`.
- `'off'` — Suppress display of the summary report.

Default: `'on'`

Output Arguments

options

Option set containing the specified options for `findop`.

Examples

Create Options Set for Operating Point Search

Create an options set for operating point search that sets the optimizer type to gradient descent and suppresses the display output of `findop`.

```
options = findopOptions('OptimizerType','graddescent',...
                        'Display','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = findopOptions;
options.OptimizerType = 'graddescent';
options.Display = 'off';
```

See Also

`| findop`

Purpose

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:

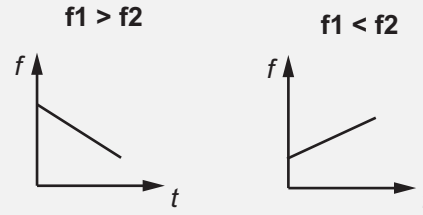
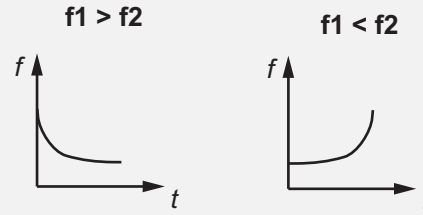
- `'FreqRange'` are the frequencies at which the linear system has interesting dynamics.
- `'Ts'` is set to avoid aliasing such that the Nyquist frequency of the signal is five times the upper end of the frequency range.
- `'NumSamples'` is set such that the frequency response estimation includes the lower end of the frequency range.

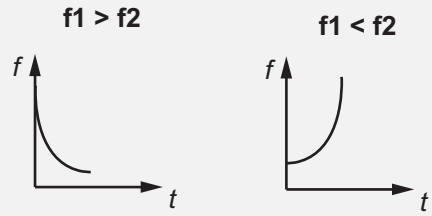
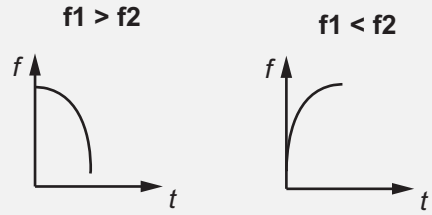
Other chirp options have default values.

'OptionName',OptionValue

Signal characteristics, specified as comma-separated pairs of option name string and the option value.

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:** $\frac{2\pi}{5 * \max(FreqRange)}$
`'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.  • `'logarithmic'`—Specifies the instantaneous frequency sweep $f_i(t)$ given by $f_i(t) = f_0 \times \beta^t \quad where \quad \beta = \left(\frac{f_1}{f_0}\right)^{\frac{1}{t_f}}$  • `'quadratic'`—Specifies the instantaneous frequency sweep $f_i(t)$:

Option Name	Option Value
	$f_i(t) = f_0 + \beta t^2 \quad where \quad \beta = (f_1 - f_0)/t_i^2$ Also specify the shape of the quadratic using the `'Shape'` option.
`'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.  • `'convex'`—Convex quadratic sweeping shape. 
`'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.Sinestream` | `frest.Random` | `frestimate` | `generateTimeseries` | `getSimulationTime`

How To

- "Creating Input Signals for Estimation" on page 3-9
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28

frest.createFixedTsSinestream

Purpose

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`

How To

- "Creating Input Signals for Estimation" on page 3-9
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28

frest.createStep

Purpose

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 pairs of option name string and the option value.

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`
`'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`

How To

- "Creating Input Signals for Estimation" on page 3-9
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28

frest.findDepend

Purpose

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. *`dirs`* is a cell array, where each element is a path string. *`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`

How To

- "Speeding Up Estimation Using Parallel Computing" on page 3-83
- "Scope of Dependency Analysis"

frest.findSources

Purpose

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`.

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`.

Input Arguments

model

String containing the name, in single quotes, 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 7-45 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`.

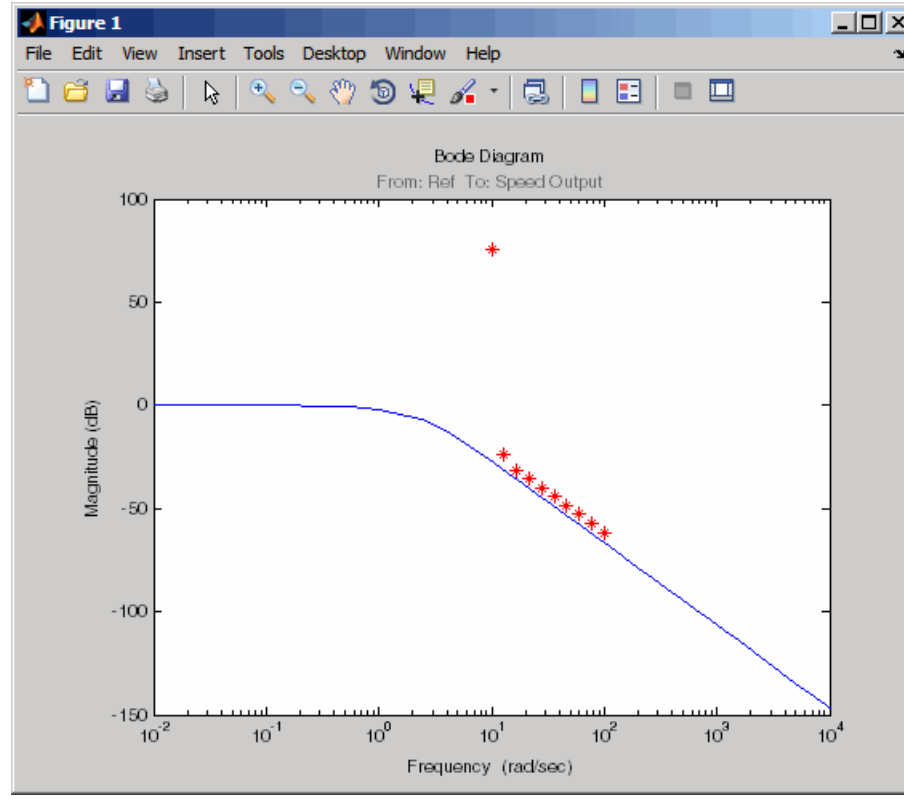
```
io = getlinio(mdl)
```

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

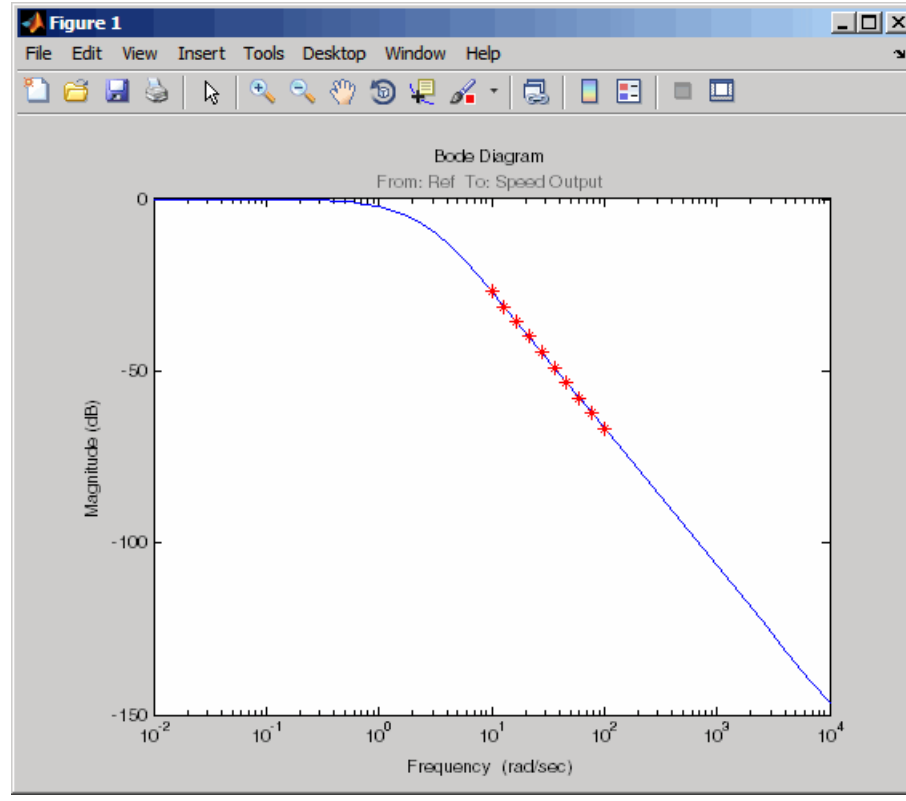
```
ans =

  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*')
```



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.

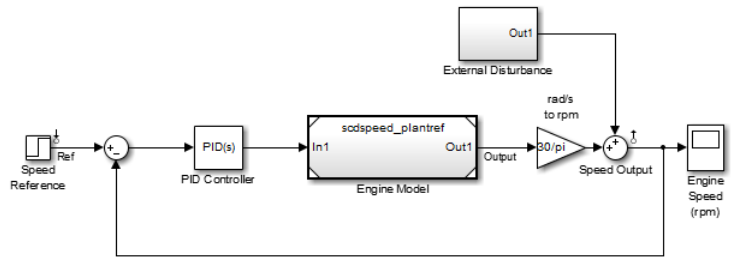
The `frest.findSources` algorithm traces every signal path that can affect the signal value at each linearization output point in the model. The paths traced include:

- 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');
```

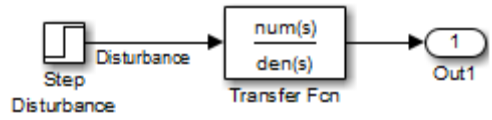


You can now run `frest.findSources` to identify the time-varying source blocks using the linearization output point defined in the model.

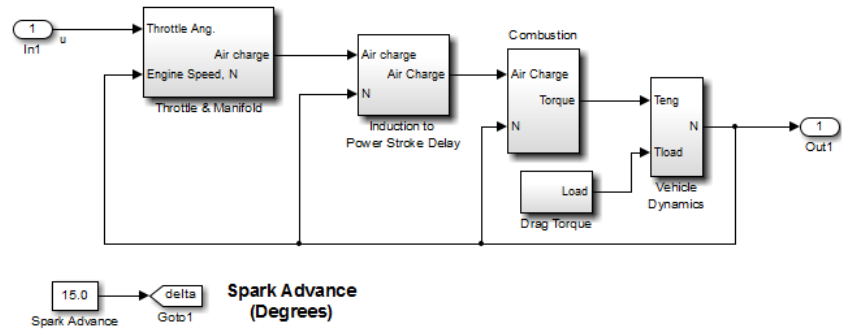
```
srcblks = frest.findSources('scdspeed_ctrlloop');
```

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 Consulting the Model Advisor in the *Simulink User's Guide*.

See Also

`frestimate | frestimateOptions`

How To

- "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 3-63

Purpose

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 timeseries 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 pairs of option name string and the option value.

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.Sinestream` | `frest.Random` | `frestimate` | `generateTimeseries` | `getSimulationTime`

How To

- "Creating Input Signals for Estimation" on page 3-9
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28

frest.simCompare

Purpose

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`.

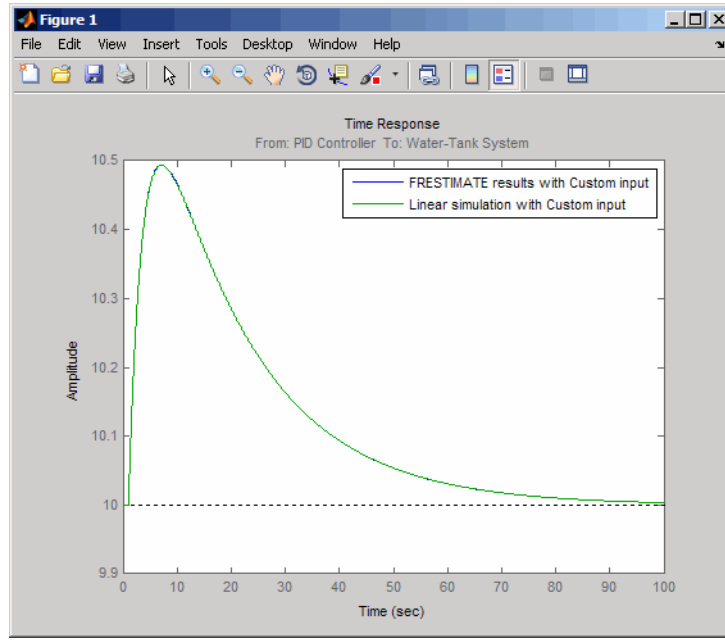
Examples

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

`frestimate` | `frest.simView`

Purpose

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`.

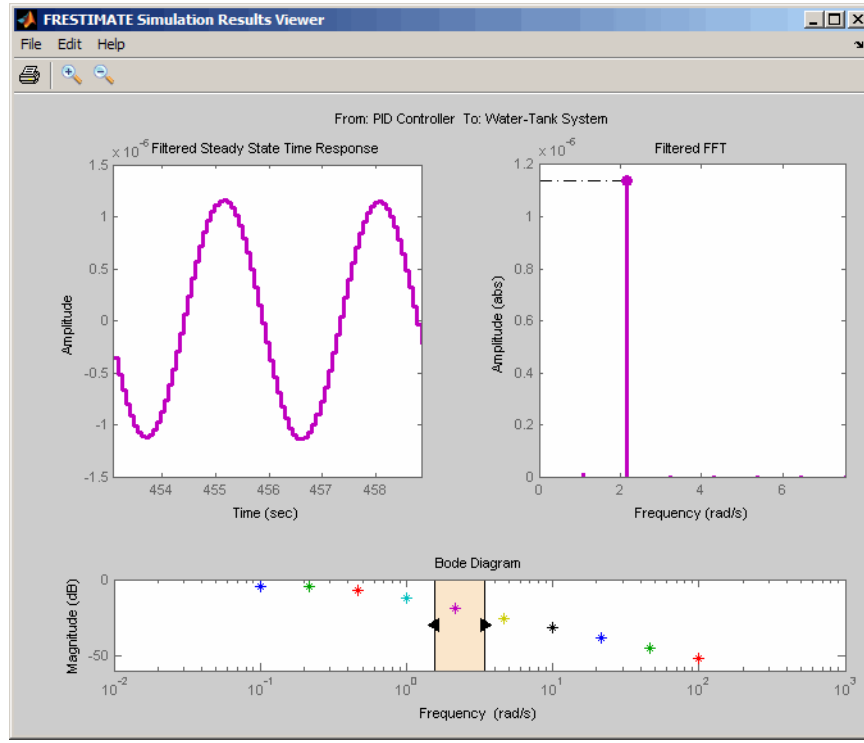
`frest.simView(simout,input,sysest,sys)` includes the linear system `sys` in the Bode plot when you create the input signal using `frest.Sinestream` or `frest.Chirp`. Use this syntax to compare the linear system to the frequency response estimation results.

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

`frestimate` | `frest.simCompare`

How To

- “Analyzing Estimated Frequency Response” on page 3-41

- “Troubleshooting Frequency Response Estimation” on page 3-50

frest.Sinestream

Purpose

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:

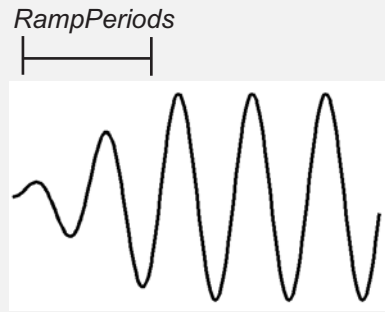
- `'Frequency'` are the frequencies at which the linear system has interesting dynamics.
- `'SettlingPeriods'` is the number of periods it takes the system to reach steady state at each frequency in `'Frequency'`.
- `'NumPeriods'` is (3 + `SettlingPeriods`) to ensure that each frequency excites the system 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.

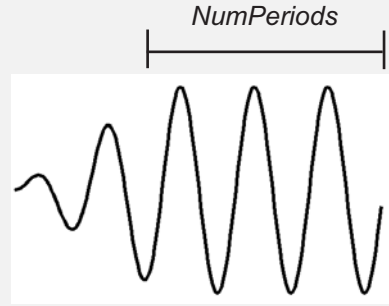
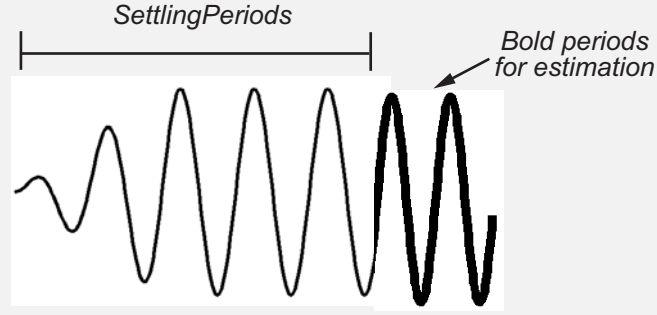
Other sinestream options have default values.

'OptionName',OptionValue

Signal characteristics, specified as comma-separated pairs of option name string and the option value.

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` 

Option Name	Option Value
`'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.
`'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 3-83.

Examples

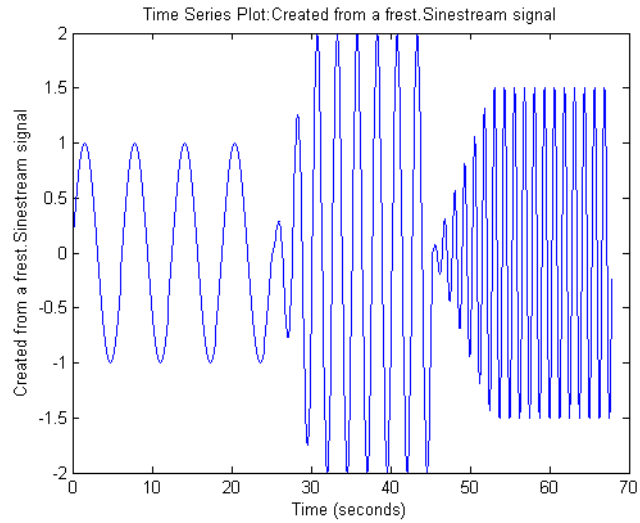
Create a sinestream signal having several different frequencies. For each frequency, specify an amplitude, a number of periods at maximum amplitude, a ramp-up period, and a number of settling periods.

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` | `frestimate` | `generateTimeseries` | `frest.createFixedTsSinestream` | `getSimulationTime`

How To

- "Creating Input Signals for Estimation" on page 3-9
- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28
- "Speeding Up Estimation Using Parallel Computing" on page 3-83

Purpose

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 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.

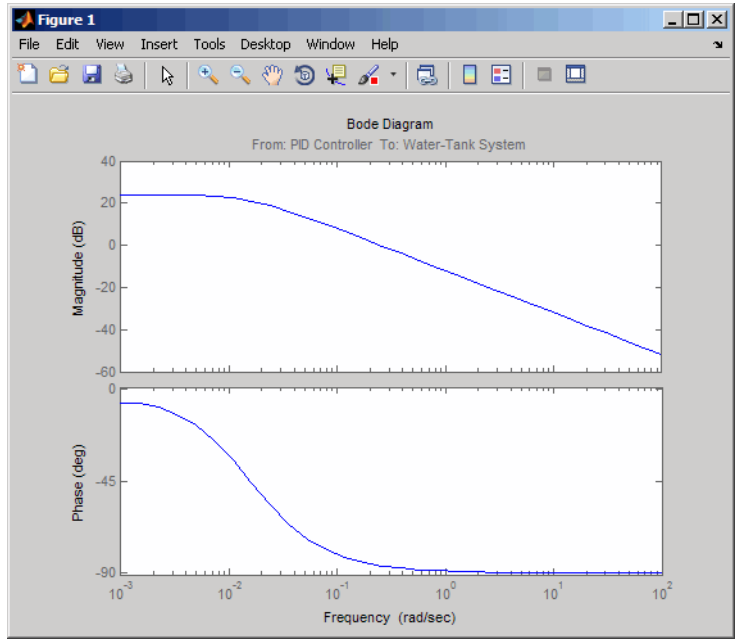
`[sysest,simout] = frestimate(model,op,io,input,options)` uses the frequency response options (`options`) to estimate the frequency response. Specify these options using `frestimateOptions`.

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:
```

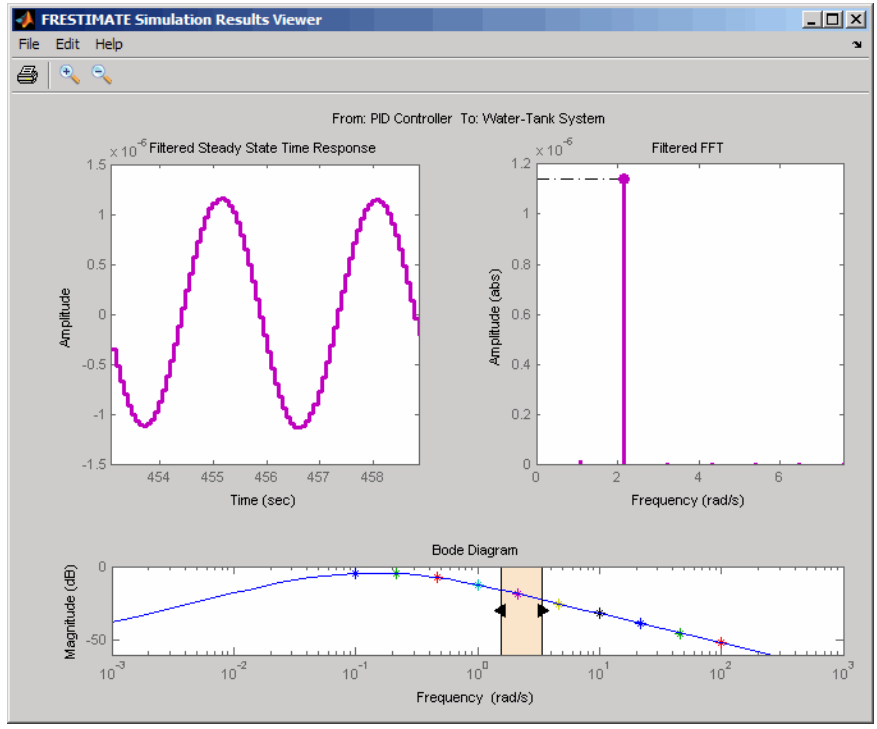
```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

% 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)
```



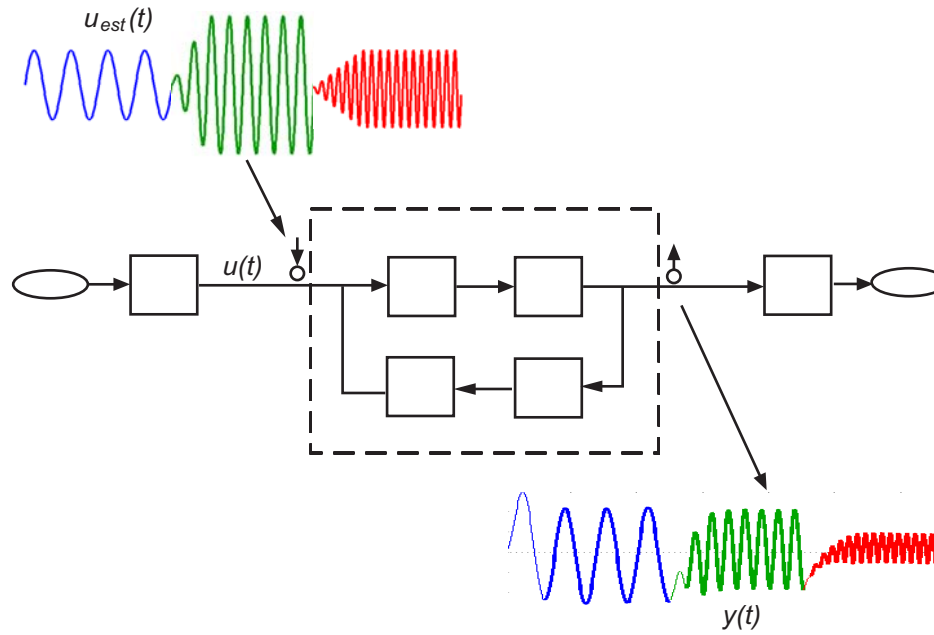
Algorithms

`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.

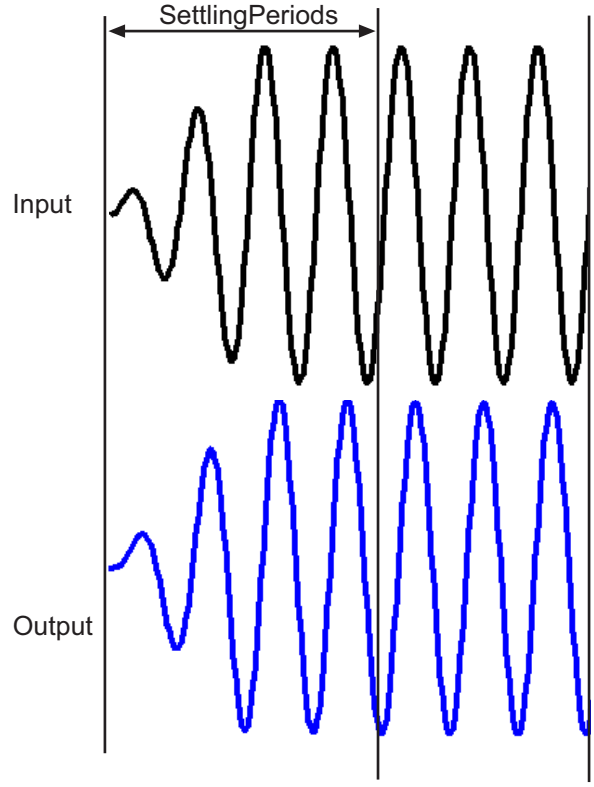
2 Simulates the output at the linearization output point.

`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.

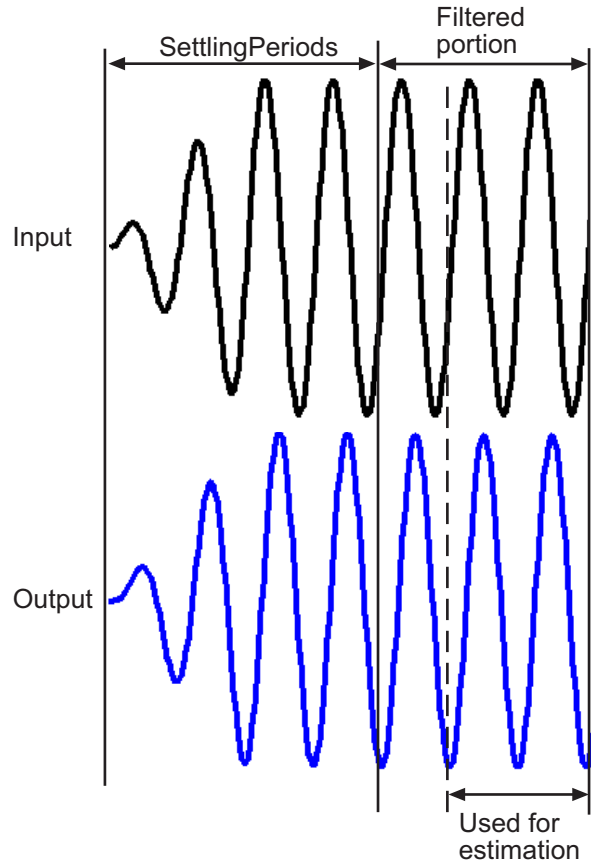
The simulated output at each frequency has a transient portion and steady state portion. `SettlingPeriods` corresponds to the transient components of the output and input signals. The periods following `SettlingPeriods` are considered to be at steady state.



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 $u_{est}(t)$:

$$Frequency\ \mathrm{Re}sponse\ Model = \frac{fft\ of\ y_{est}(t)}{fft\ of\ u_{est}(t)}$$

To compute the response at each frequency, `frestimate` uses only the simulation output at that frequency.

See Also

`frest.Sinestream` | `frest.Chirp` | `frest.Random` | `frest.simView` | `frestimateOptions` | `getSimulationTime`

How To

- "Estimate Frequency Response (MATLAB Code)" on page 3-37
- "Estimate Frequency Response Using Linear Analysis Tool" on page 3-28
- "Speeding Up Estimation Using Parallel Computing" on page 3-83

frestimateOptions

Purpose

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 pairs of option name string and the option value.

Option Name	Option Value
`'BlocksToHoldConstant'`	An array of `Simulink.BlockPath` that specifies the paths of time-varying source blocks to hold constant during frequency response estimation. Use `frest.findSources` to identify time-varying source blocks that can interfere with frequency response estimation. **Default**: empty
`'UseParallel'`	Set to `'on'` to enable parallel computing for estimations with the `frestimate` command. **Default**: `'off'`
`'ParallelPathDependencies'`	A cell array of strings 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

`frestimate` | `frest.findSources`

Purpose

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

`frestimate` | `frest.Sinestream` | `fdel`

How To

- “Time Response Not at Steady State” on page 3-50

generateTimeseries

Purpose

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

`frestimate` | `frest.Sinestream` | `frest.Chirp` | `frest.Random`

Purpose

Properties of linearization I/Os and operating points

Syntax

```
get(ob)
get(ob,'PropertyName')
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

getinputstruct

Purpose

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`

Purpose

Linearization input/output (I/O) settings for Simulink model, Linear Analysis Plots or Model Verification block

Syntax

```
io = getlinio('sys')
io = getlinio('blockpath')
```

Alternatives

As an alternative to `getlinio`, view linearization I/Os annotated in the Simulink model in the:

- **Exact Linearization** tab of the Linear Analysis Tool. In the **Setup** section, click to view and edit the linearization I/Os. The icon appears only when **Analysis I/Os** is set to `Model I/Os`.
- **Linearization inputs/outputs** table in the **Linearizations** tab of the Block Parameters dialog box for Linear Analysis Plots or Model Verification blocks.

Description

`io = getlinio('sys')` finds all linearization inputs/outputs (I/Os) in the Simulink model, `sys`, and returns a vector of objects, `io`. Each object represents a linearization annotation in the model and is associated with an output port of a Simulink block. Before running `getlinio`, use the right-click menu to insert the linearization annotations, or I/Os, on the signal lines of the model diagram.

`io = getlinio('blockpath')` finds all I/Os in a Linear Analysis Plots block or a Model Verification block. `blockpath` is the full path to the block. `io` is a vector of objects and has an entry for each linearization port used by the block.

Each object within the vector, `io`, has the following properties:

`Active`	Set this value to `'on'`, when the I/O is used for linearization, and `'off'` otherwise
`Block`	Name of the block with which the I/O is associated
`PortNumber`	Integer referring to the output port with which the I/O is associated

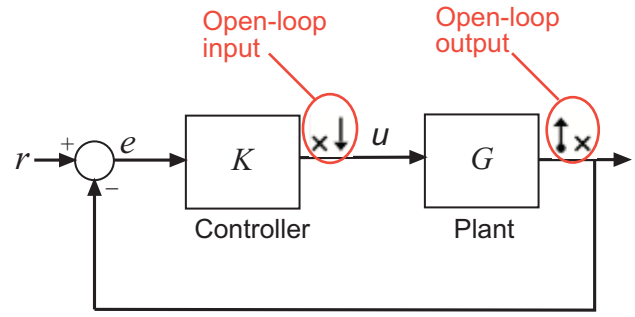
`Type`

Choose one of the following linearization I/O types:

- `'openinput'` — Open-loop input. Specifies a linearization input point after a loop opening.

  Typically, you use this input type with an open-loop linearization output to linearize a plant or controller.

  For example, to compute the plant transfer function, `G`, in the following feedback loop, specify the linearization points as shown:



  Similarly, you can compute the controller transfer function, `K`, by specifying `openinput` at the input signal and open-loop linearization output at the output signal of the Controller block.

- `'openoutput'` — Open-loop output. Specifies a linearization output point before a loop opening.

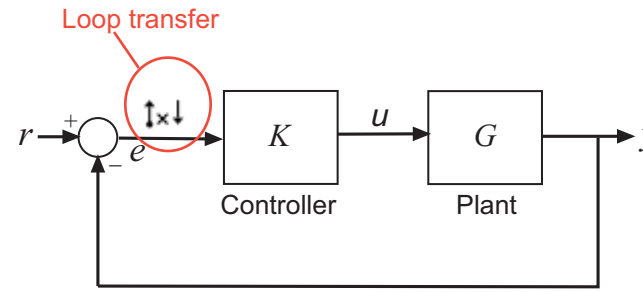
  Typically, you use this output type with an open-loop linearization input `openinput` or input perturbation `input` to linearize

a plant or controller, as shown in the preceding figure.

- `'looptransfer'` — Loop transfer. Specifies an output point before a loop opening followed by an input.

  Use this input/output type to compute the open-loop transfer function around the loop.

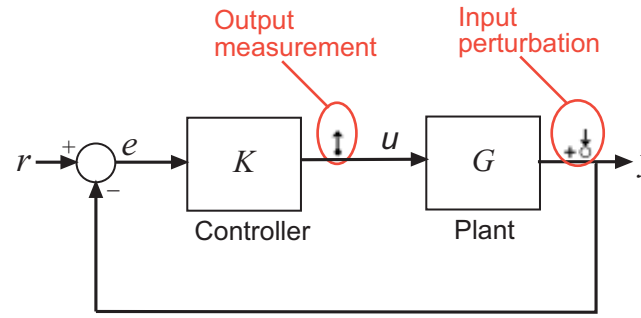
  For example, to compute `-GK` in the following feedback loop, specify the linearization input/output point as shown:



  Similarly, compute `-KG` by specifying `looptransfer` at the output signal of the Controller block.

- `'input'` — Input perturbation. Specifies an additive input to a signal.

  For example, to compute the response `-K/(1+KG)` of the following feedback loop, specify an input perturbation and an output measurement point as shown:



Similarly, you can compute `G/(1+GK)` using `input` at the output signal of the Controller block and an output measurement `output` at the output signal of the Plant block.

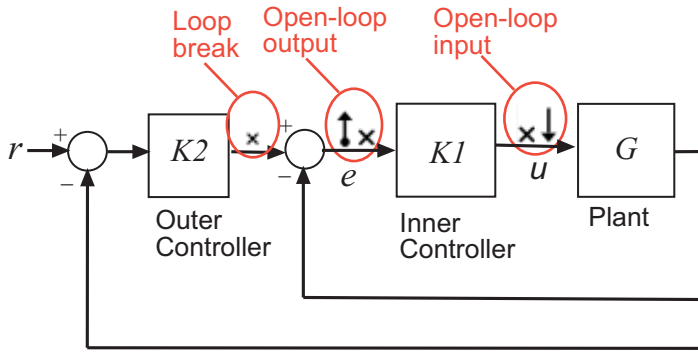
- `'output'` — Output measurement. Takes measurement at a signal.

  For example, to compute the response `-K/(1+KG)`, specify an output measurement point and an input perturbation as shown in the preceding figure.

- `'loopbreak'` — Loop break. Specifies a loop opening.

  Use to compute open-loop transfer function around a loop. Typically, you use this input/output type when you have nested loops or want to ignore the effect of some loops.

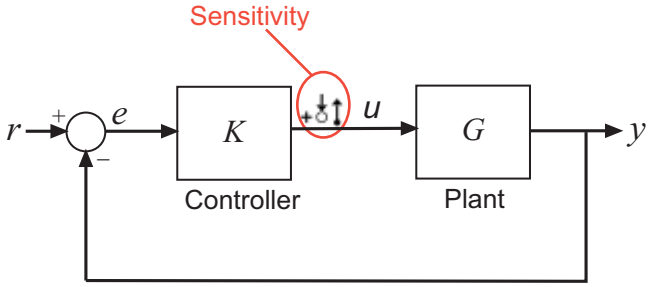
  For example, to compute the inner loop seen by `K1` and exclude the outer loop, specify the input/output points and `loopbreak` as shown:



- `'sensitivity'` — Sensitivity. Specifies an additive input followed by an output measurement.

  Use to compute sensitivity transfer function for an additive disturbance at the signal.

  For example, compute the input/load sensitivity, `1/(1+KG)`, in the following feedback loop, specify the linearization input/output point as shown:

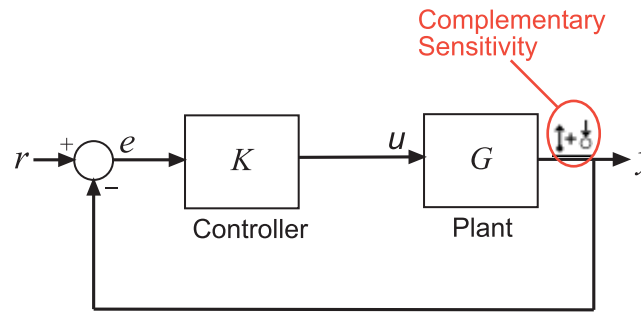


  Similarly, compute output sensitivity at the plant output, `1/(1+GK)`, by specifying a `sensitivity` input/output point at the output signal of the Plant block.

- `'compsensitivity'` — Complementary sensitivity. Specifies an output followed by an additive input.

  Use to compute closed-loop transfer function around the loop.

  For example, to compute `-GK/(1+GK)` (the transfer function from `r` to `y`) in the following feedback loop, specify the linearization input/output point at the output signal of the Plant block as shown:



`BusElement`	Bus element name with which the I/O is associated. Empty string (`''`) if the I/O is not a bus element.
`Description`	String description of the I/O object

You can edit this I/O object to change its properties. Alternatively, you can change the properties of `io` using the `set` function. To upload an edited I/O object to the Simulink model diagram, use the `setlinio` function. Use I/O objects with the function `linearize` to create linear models.

Examples

Find linearization inputs/outputs in a Simulink model.

Before creating a vector of I/O objects using `getlinio`, you must add linearization annotations representing the I/Os, such as input points or output points, to a Simulink model.

1. Open a Simulink model.

   ```
   magball
   ```

2. Right-click the signal line between the Magnetic Ball Plant and the Controller. Select **Linear Analysis Points > Input Perturbation** from the menu to place an input point on this signal line.

   A small arrow pointing toward a small circle just above the signal line represents the input point. The input point is not the output of the block, rather it is an additive input to the signal.

3. Right-click the signal line after the Magnetic Ball Plant. Select **Linear Analysis Points > Open-loop Output** to place an output point on this signal line.

   A small arrow pointing away from the signal line represents the output point.

4. Create a vector of I/O objects for this model.

   ```
   io=getlinio('magball')
   ```

   This syntax returns a formatted display of the linearization I/Os.

   ```
   2x1 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
```

`io` is a vector with two entries representing the two linearization annotations previously set in the model diagram. MATLAB also displays:

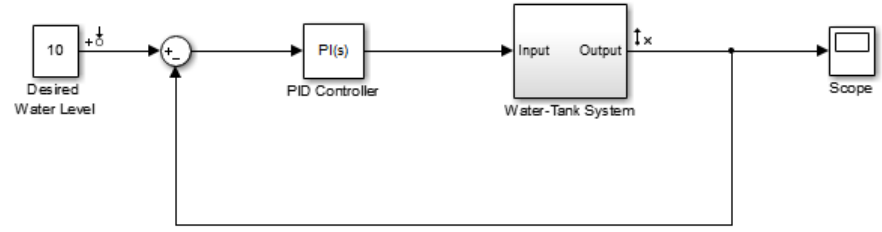
- The linearization I/O type (input or output) and whether the IO is a loop opening
- Block name associated with the I/O
- Port number associated with the I/O

Display the properties of each I/O object using the `get` function.

This example shows how to find linearization inputs/outputs in a Linear Analysis Plots block to update the I/Os.

1. Open the `watertank` model, and specify input and output (I/O).
   a. Right-click the Desired Water Level output signal, and select **Linear Analysis Points > Input Perturbation**.
   b. Right-click the Water-Tank System output signal, and select **Linear Analysis Points > Open-loop Output**.

   The linearization I/O markers appear in the model, as shown in the next figure.



Alternatively, you can use `linio`.

2 Drag and drop a Bode Plot block from the Simulink Control Design Linear Analysis Plots library into the Simulink Editor. When you drag and drop the block, the block I/Os are set to the model I/Os.

3 Find all I/Os used by the Bode Plot block.

```
io = getlinio('watertank/Bode Plot')
```

The following results appear at the MATLAB prompt:

```
2x1 vector of Linearization IOs:
-------------------------
1. Linearization input perturbation located at the following signal:
- Block: watertank/Desired  Water Level
- Port: 1

2. Linearization open-loop output located at the following signal:
- Block: watertank/Water-Tank System
- Port: 1
```

See Also

`get` | `linearize` | `linio` | `set` | `setlinio`

getlinplant

Purpose

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

getSimulationTime

Purpose

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

Simulation Time for Frequency Response Estimation

Create a sinestream input signal and calculate the final simulation time of an estimation using that signal.

```
input = frest.Sinestream('Amplitude',1e-3,...
                         'Frequency',logspace(1,3,50),...
                         'SamplesPerPeriod',40,'FreqUnits','Hz');
tfinal = getSimulationTime(input)
```

```
tfinal =

    4.4186
```

The sinestream signal `input` includes 50 frequencies spaced logarithmically between 10 Hz and 1000 Hz. Each frequency is sampled 40 times per period.

The resulting `tfinal` indicates that frequency response estimation of any model with this input signal would simulate the model for 4.4186 s.

See Also

`frestimate | frest.Sinestream | frest.Chirp | frest.Random`

Related Examples

- "Creating Sinestream Input Signals" on page 3-9
- "Creating Chirp Input Signals" on page 3-19

Concepts

- "Ways to Speed up Frequency Response Estimation" on page 3-81

Purpose

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

getxu

Purpose

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`

Purpose

Initialize operating point specification values

Syntax

```
opnew=initopspec(opspec,oppoint)
opnew=initopspec(opspec,x,u)
opnew=initopspec(opspec,xstruct,u)
```

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 1-36.

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.

See Also

`findop` | `getstatestruct` | `getxu` | `operpoint` | `operspec`

linearize

Purpose

Linear approximation of Simulink model or block

Syntax

```
linsys = linearize(sys,io)
linsys = linearize(sys,op)
linsys = linearize(sys,op,io)
linsys = linearize(sys,op,io,options)
linsys = linearize(sys,io,param)
[linsys,op] = linearize(sys,io,tsnapshot)
linsys = linearize(sys,op,io,'StateOrder',stateorder)
linblock = linearize(sys,blockpath,op)
linsys = linearize(sys,blocksub,op,io)
```

Description

`linsys = linearize(sys,io)` linearizes the nonlinear Simulink model defined by the linearization I/O points `io`. Linearization uses the operating point that corresponds to the initial states and input levels in the Simulink model.

`linsys = linearize(sys,op)` linearizes the entire Simulink model such that the linearization I/O points are the root-level inport and output blocks in `sys`. Linearization uses the operating point `op`.

`linsys = linearize(sys,op,io)` linearizes the model specified by linearization I/O points `io`.

`linsys = linearize(sys,op,io,options)` uses algorithm options specified in `options`.

`linsys = linearize(sys,io,param)` batch linearizes the model using the specified I/O points, varying the values of the parameters specified by `param`. Linearization uses the operating point that corresponds to the initial states and input levels in the Simulink model.

`[linsys,op] = linearize(sys,io,tsnapshot)` linearizes the model at one or more simulation times `tsnapshot`. Returns the operating point `op` that corresponds to the simulation snapshot. Omit `io` when you want to use the root-level inport and output blocks in `sys` as linearization I/O points.

`linsys = linearize(sys,op,io,'StateOrder',stateorder)` returns a linear model with a specified state order.

`linblock = linearize(sys,blockpath,op)` linearizes the block in the model `sys` specified by the `blockpath`. Linearization uses the operating point `op`.

`linsys = linearize(sys,blocksub,op,io)` linearizes the nonlinear Simulink model defined by the linearization I/O points `io`. `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. Omit the operating point `op` when you want to use the model operating point. Omit `io` when you want to use the root-level inport and output blocks in `sys` as linearization I/O points.

Input Arguments

sys

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

io

Linearization I/O object, specified using `linio`.

If the linearization input and output points are annotated in the Simulink model, extract these points from the model into `io` using `getlinio`.

`io` must correspond to the Simulink model `sys` or some normal mode model reference in the model hierarchy.

op

Operating point object, specified using `operpoint` or `findop`.

`op` must correspond to the Simulink model `sys`.

options

Algorithm options, specified using `linearizeOptions`.

param

Parameter variations, specified as:

- Structure — For a single parameter, `param` must be a structure with the following fields:
  - `Name` — Parameter name, specified as a string or MATLAB expression
  - `Value` — Parameter sample values, specified as a double array

  For example:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, suppose you want to vary the value of the `A` and `b` model parameters 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, the software batch linearizes the model using a single model compilation.

tsnapshot

Simulation snapshot time instants when to linearize the model, specified as a scalar or vector.

stateorder

State order in linearization results, specified as a cell array of block paths for blocks with states. Each block path is string of the form *model/subsystem/block* that uniquely identifies a block in the model.

The order of the block paths in the cell array should match the desired order of the linearized model states.

blockpath

Block to linearize, specified as a full block path. A block path is string of the form *model/subsystem/block* that uniquely identifies a block in the model.

blocksub

Substitute linearizations for blocks and model subsystems. Use `blocksub` to specify a custom linearization for a block or subsystem. You also can use `blocksub` for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

`blocksub` is an *n*-by-1 structure, where *n* is the number of blocks for which you specify the linearization. `blocksub` has these fields:

- `Name` — Block path corresponding to the block for which you want to specify the linearization.

  `blocksub.Name` is a string of the form *model/subsystem/block* that uniquely identifies a block in the model.

- `Value` — Desired linearization of the block, specified as a structure with these fields:

  `Specification` — Block linearization, specified as a string. The string can include a MATLAB expression or function that returns one of the following:

  - Linear model in the form of a D-matrix
  - Control System Toolbox LTI model object
  - Robust Control Toolbox uncertain state space or uncertain real object (requires Robust Control Toolbox software)

  If `blocksub.Value.Specification` is a MATLAB expression, this expression must follow the resolution rules , as described in “Symbol Resolution”.

If `blocksub.Value.Specification` is a function, this function must have one input argument, `BlockData`, which is a structure that the software creates automatically and passes to the specification function. `BlockData` has the following fields:

- `BlockName` is the name of the Simulink block with the specified linearization.
- `Parameters` is a structure array containing the evaluated values for the block. Each element of the array has the fields `'Name'` and `'Value'`, which contain the name and evaluated value, respectively, for the parameter.
- `Inputs` is a structure that has the following fields:
  - `BlockName` — Contains the name of the block whose output connects to the input of the block whose linearization you are specifying. For example, if you specify the linearization of a block called `Dynamics`, and the second input of `Dynamics` is driven by a signal from a block called `Torque`, then `BlockData.Inputs(2).BlockName` is the full block path name of `Torque`.
  - `PortIndex` — Identifies which output port of `BlockName` corresponds to the input of the block whose linearization you are specifying. For example, if the third output from `Torque` drives the second input of `Dynamics`, then `BlockData.Inputs(2).PortIndex = 3`.
  - `Values` — The value of the signal specified by `BlockName` and `PortIndex`. If this signal is a vector-valued signal, `Values` is a vector of corresponding dimension.
- `ny` is the number of output channels of the block linearization.
- `nu` is the number of input channels of the block linearization.
- `Type` — Specification type, specified as one of these strings:

```
'Expression'
'Function'
```

- `ParameterNames` — Linearization function parameter names, specified as a comma-separated list of strings. Specify only when `blocksub.Value.Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block

  You also must specify the corresponding `blocksub.Value.ParameterValues` field.

- `ParameterValues` — Linearization function parameter values that correspond to `blocksub.Values.ParameterNames`. Specify only when `blocksub.Value.Type = 'Function'`.

  `blocksub.Value.ParameterValues` is a comma separated list of values. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`.

- `BlockLinearization` is a state-space (`ss`) model that is the current default linearization of the block. You can use `BlockData.BlockLinearization` in the specification function to specify a block linearization that depends on the default linearization, such as the default linearization multiplied by a time delay.

Output Arguments

linsys

Linear time-invariant state-space model that approximates the nonlinear model specified by linearization I/O points `io`.

`linsys` is returned as an `ss` object.

op

Operating point corresponding the simulation snapshot of the states and input levels at `tsnapshot`, returned as an operating point object. This is the same object as returned using `operpoint` or `findop`.

View `op` values to determine whether the model was linearized at a reasonable operating point.

linblock

Linear time-invariant state-space model that approximates the specified nonlinear block, returned as an `ss` object.

Examples

Linearization at Model Operating Point

This example shows how to use `linearize` to linearize a model at the operating point specified in the model. The model operating point consists of the model initial state values and input signals.

1 Open Simulink model.

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

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

2 Specify to linearize the Water-Tank System block using linearization I/O points.

```
sys_io(1)=linio('watertank/PID Controller',1,'input');
sys_io(2)=linio('watertank/Water-Tank System',1,'openoutput');
```

Each linearization I/O point is associated with a block outport. For example, to specify a linearization input point at the Water-Tank System block input, you must associate this input point with the outport of the PID Controller block.

`sys_io` is an object array that includes two linearization I/O objects. The first object, `sys_io(1)`, specifies the linearization input point on the first `watertank/PID Controller` output signal. The second object, `sys_io(2)`, specifies the linearization output point on the first `watertank/Water-Tank System` output signal.

Note When there are multiple block output signals and you want to specify an output port other than the first output, enter the desired output port number as the second argument of `linio`.

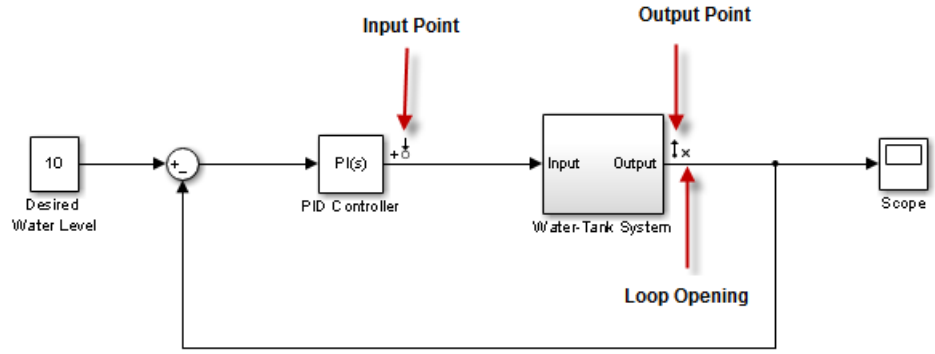
Specifying the linearization output point as open loop removes the effects of the feedback signal on the linearization without changing the model operating point.

Note Do not open the loop by manually removing the feedback signal from the model. Removing the signal manually changes the operating point of the model.

3 Update the model to reflect the modified linearization I/O object.

```
setlinio(sys,sys_io);
```

When you add input points, output points, or loop openings, linearization I/O markers appear in the model. Use these to visualize your linearization points.



4 Linearize the Water-Tank System block at the model operating point.

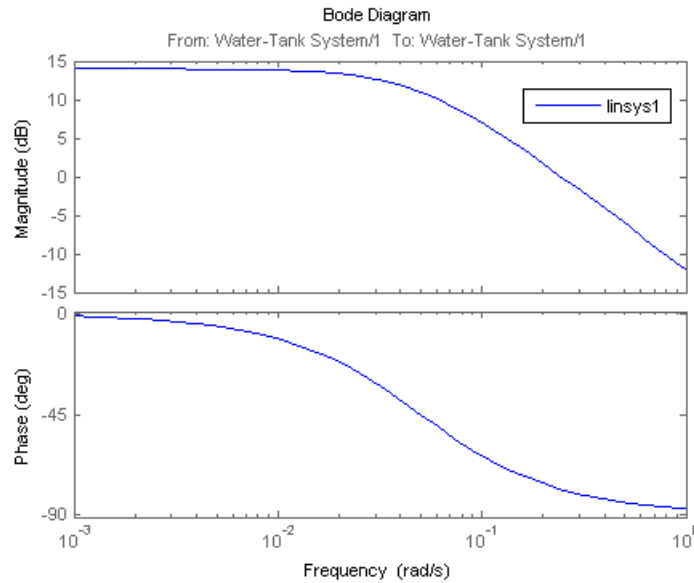
```
linsys = linearize(sys,sys_io);
bdclose(sys);
```

`linsys` is a state-space model object.

5 Plot a Bode plot of the linearized model.

```
bode(linsys)
```

The resulting Bode plot looks like a stable first-order response, as expected.



Linearization at Simulation Snapshot

This example shows how to use `linearize` to linearize a model by simulating the model and extracting the state and input levels of the system at specified simulation times.

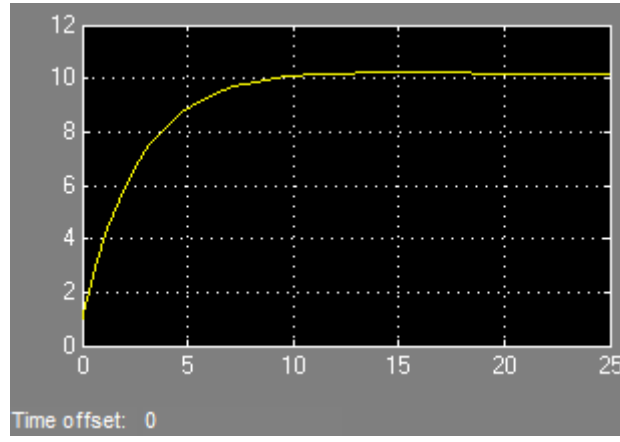
1 Open Simulink model.

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

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

2 Simulate the model to determine the time when the model reaches steady state.

The Scope block shows that the system reaches steady state at approximately 20 time units.



3 Specify to linearize the open-loop Water-Tank System.

```
sys_io(1)=linio('watertank/PID Controller',1,'input');
sys_io(2)=linio('watertank/Water-Tank System',1,'openoutput');
```

The last input argument for computing `sys_io(2)` opens the feedback loop.

Note Do not open the loop by manually removing the feedback signal from the model. Removing the signal manually changes the operating point of the model.

---

4 Linearize the Water-Tank System block at a simulation time of 20time units.

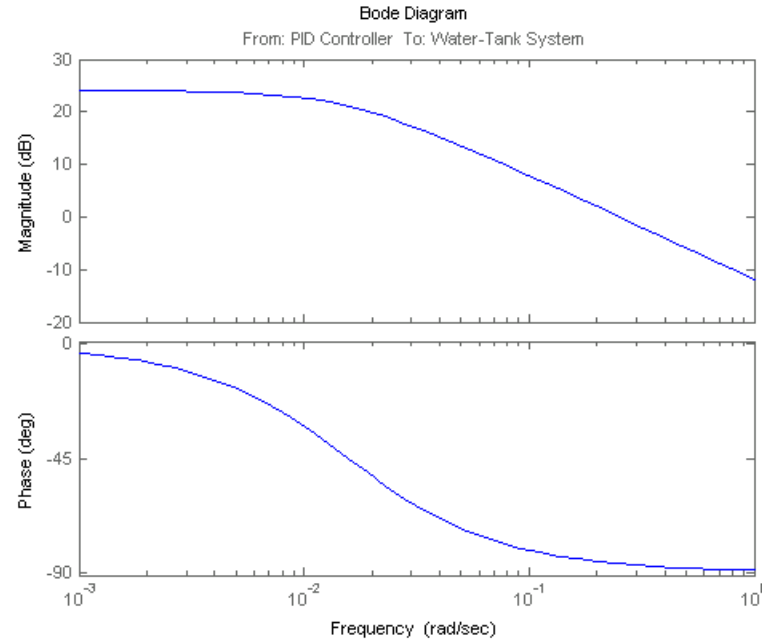
```
tsnapshot = 20;
linsys = linearize(sys,sys_io,tsnapshot);
bdclose(sys);
```

`linsys` is a state-space model object.

5 Plot a Bode plot of the linearized model.

```
bode(linsys)
```

The resulting Bode plot looks like a stable first-order response, as expected.



Linearization at Trimmed Operating Point

This example shows how to use `linearize` to linearize a model using a trimmed operating point.

1 Open Simulink model.

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

2 Create operating point specification object.

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

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

3. Find the steady-state operating point using trim analysis.

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

4. Specify to linearize the open-loop Water-Tank System.

```
sys_io(1)=linio('watertank/PID Controller',1,'input');
sys_io(2)=linio('watertank/Water-Tank System',1,'openoutput');
```

5. Linearize the Water-Tank System block at the trimmed operating point.

```
linsys = linearize(sys,op,sys_io);
bdclose(sys);
```

`linsys` is a state-space model object.

Linearization at Multiple Simulation Snapshots

This example shows how to use `linearize` to linearize a model at multiple simulation snapshots.

1. Open Simulink model.

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

2. Specify to linearize the open-loop Water-Tank System.

```
sys_io(1)=linio('watertank/PID Controller',1,'input');
sys_io(2)=linio('watertank/Water-Tank System',1,'openoutput');
```

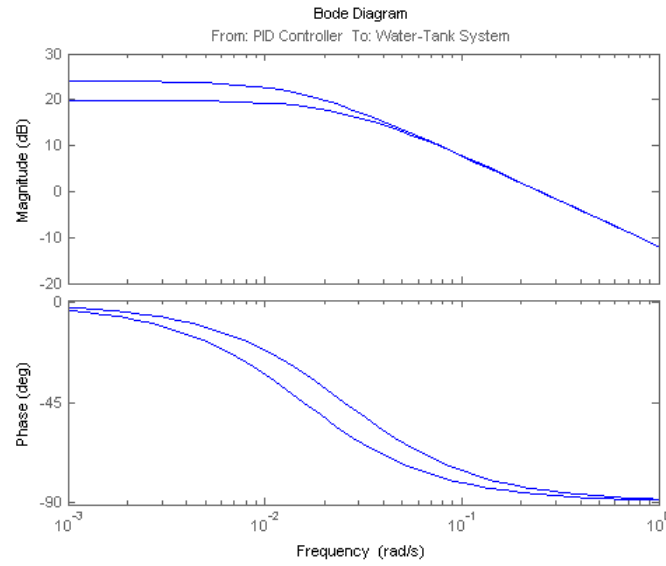
3. Define the simulation times at which to linearize the model.

```
tsnapshot = [1,20];
linsys = linearize(sys,sys_io,tsnapshot);
```

```
bdclose(sys);
```

4 Plot the Bode response.

```
bode(linsys);
```



Plant Linearization at Model Operating Point

This example shows how to use `linearize` to linearize a subsystem at the model operating point.

Use this approach instead of defining linearization I/O points when the plant is a subsystem or a block.

1 Open Simulink model.

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

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

2 Linearize the Water-Tank System block.

```
linsys = linearize(sys,blockpath);
bdclose(sys);
```

Algorithms

By default, `linearize` automatically sets the Simulink model properties:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`

After the linearization completes, Simulink restores the original model properties.

Alternatives

As an alternative to the `linearize` function, linearize models using:

- The Linear Analysis Tool. For example, see "Linearize at Model Operating Point" on page 2-43.
- The `slLinearizer` interface. For example, see "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169.

See Also

`findop` | `linlftfold` | `linearizeOptions` | `slLinearizer`

Related Examples

- "Linearize at Model Operating Point" on page 2-43
- "Linearize at Trimmed Operating Point" on page 2-57
- "Batch Linearize Model for Parameter Value Variations Using linearize" on page 2-165

Concepts

- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

Purpose

Set options for linearization

Syntax

```
options = linearizeOptions
options = linearizeOptions(Name,Value)
```

Alternatives

As an alternative to `linearizeOptions` function, set options for linearization in the Linear Analysis Tool.

Description

`options = linearizeOptions` returns the default linearization options.

`options = linearizeOptions(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 commands that perform linearization, including `linearize`, `slLinearize`, `slTunable`, `ulinearize`, and `linlft`.

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`.

For example, `'RateConversionMethod','tustin'` sets the `'RateConversionMethod'` option to the value `'tustin'`.

`linearizeOptions` takes the following `Name` arguments:

'LinearizationAlgorithm'

Algorithm used for linearization, specified as either `'blockbyblock'` or `'numericalpert'`.

- `'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 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.
- Nonminimal states are automatically removed.
- You can specify that linearization include uncertainty (requires Robust Control Toolbox software).

Default: `'blockbyblock'`

'SampleTime'

The interval of time at which the signal is sampled, specified as a scalar value:

- –1 to use the longest sample time that contributes to the linearized model
- 0 for continuous-time systems
- Positive scalar value for discrete-time systems

Default: –1

'UseFullBlockNameLabels'

Flag indicating whether to truncate names of I/Os and states in the linearized model, specified as either `'off'` or `'on'`.

- `'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.

Default: `'off'`

'UseBusSignalLabels'

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as either `'off'` or `'on'`.

- `'off'` — Use bus signal channel number to label I/Os on bus signals in your linearization results.
- `'on'` — Use bus signal names to label I/Os on bus signals in your linearization results. 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

---

Note You cannot use this option when your model has mux/bus mixtures. For information on how to avoid buses used as muxes, see "Prevent Bus and Mux Mixtures" in the Simulink documentation.

Default: `'off'`

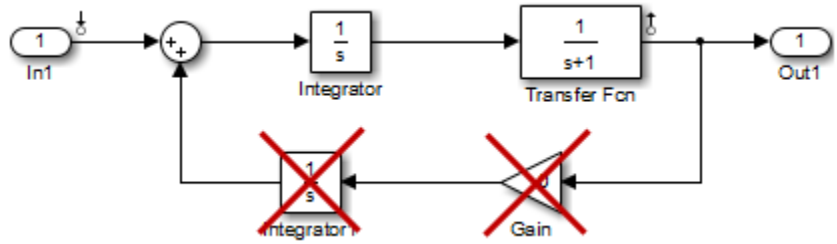
'BlockReduction'

Flag indicating whether to omit blocks that are not in the linearization path, specified as either `'off'` or `'on'`. This flag is ignored when `'LinearizationAlgorithm'` is `'numericalpert'`.

Set `'BlockReduction'` to `'on'` (default) to eliminate from the linearized model those blocks that are not in the path of the linearization. Block reduction eliminates the states of blocks in dead linearization paths from your linearization results. Some examples of dead linearization paths are linearization paths that 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, with 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).



Set `'BlockReduction'` to `'off'` to return a linearized model that includes all of the states of the model.

Default: `'on'`

'IgnoreDiscreteStates'

Flag indicating whether to remove discrete-time states from the linearization, specified as either `'off'` or `'on'`. This flag is ignored when `'LinearizationAlgorithm'` is `'numericalpert'`.

- `'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.

Default: `'off'`

'RateConversionMethod'

Method used for rate conversion when linearizing a multirate system, specified as one of the following strings:

- `'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, and examples, on methods and algorithms for rate conversions and linearization of multirate models, see:

- Linearization of Multirate Models
- Linearization Using Different Rate Conversion Methods

- "Continuous-Discrete Conversion Methods" in the Control System Toolbox documentation

This option is ignored when `'LinearizationAlgorithm'` is `'numericalpert'`.

Default: `'zoh'`

'PreWarpFreq'

Prewarp frequency in rad/s, specified as a nonnegative scalar.

Default: 0 (no prewarp)

'UseExactDelayModel'

Flag indicating whether to compute linearization with an exact delay representation, specified as either `'off'` or `'on'`. This flag is ignored when `'LinearizationAlgorithm'` is `'numericalpert'`.

- `'off'` — Return a model with approximate delays.
- `'on'` — Return a linear model with an exact delay representation.

Default: `'off'`

'NumericalPertRel'

Perturbation level for obtaining the linear model by numerical perturbation, specified as a positive scalar. This option is ignored unless `'LinearizationAlgorithm'` is `'numericalpert'`. The perturbation of the system's states is specified by:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |x|$$

The perturbation of the system's inputs is specified by:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |u|$$

Default: `1e-5`

'NumericalXPert'

Perturbation levels for the system's states, specified as an operating point object.

To set individual perturbation levels for each of the system's states:

1. Use the `operpoint` command to create an operating point object for the model.
2. Set the state values in the operating point object to the desired perturbation levels.
3. Set the value of the `'NumericalXPert'` option to the operating point object.

'NumericalUPert'

Perturbation levels for the system's inputs, specified as an operating point object.

To set individual perturbation levels for each of the system's inputs:

1. Use the `operpoint` command to create an operating point object for the model.
2. Set the input values in the operating point object to the desired perturbation levels.
3. Set the value of the `'NumericalUPert'` option to the operating point object.

Output Arguments

options

Option set containing the specified options for linearization.

Examples

Create Options Set for Linearization

Create an options set for linearization that specifies prewarp rate conversion 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';
```

See Also

`slLinearizer` | `linearize` | `slTunable` | `slTunable.linearize` | `ulinearize` | `linlft`

Purpose

Define linearization input/output (I/O) points for Simulink model

Syntax

```
io = linio('blockname', portnum)
io = linio('blockname', portnum, type)
io = linio('blockname', portnum, type, [],
'buselementname')
```

Alternatives

As an alternative to `linio`, create linearization I/O settings by using the right-click menu on the signal in the model diagram or in the Linear Analysis Tool.

Description

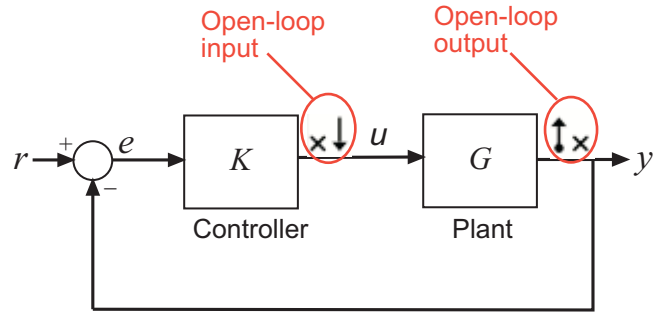
`io = linio('blockname', portnum)` creates a linearization input/output (I/O) object for the signal that originates from the outport with port number `portnum` of the block `blockname` in a Simulink model. The default I/O type is `'input'` which applies an additive input to the signal. Use `io` with `linearize` to create linearized models.

`io = linio('blockname', portnum, type)` specifies the type of linearization I/O. `type` must be one of the following strings:

- `'openinput'` — Open-loop input. Specifies a linearization input point after a loop opening.

  Typically, you use this input type with an open-loop linearization output to linearize a plant or controller.

  For example, to compute the plant transfer function, `G`, in the following feedback loop, specify the linearization points as shown:



Similarly, you can compute the controller transfer function, `K`, by specifying `openinput` at the input signal and open-loop linearization output at the output signal of the Controller block.

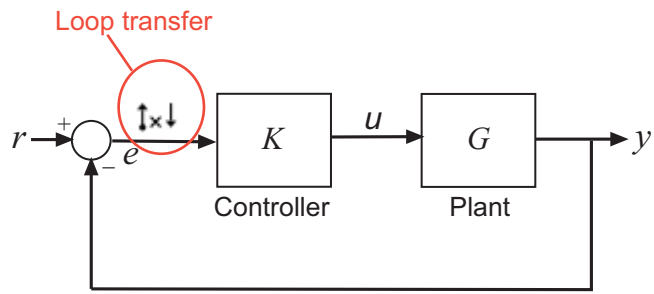
- `'openoutput'` — Open-loop output. Specifies a linearization output point before a loop opening.

  Typically, you use this output type with an open-loop linearization input `openinput` or input perturbation `input` to linearize a plant or controller, as shown in the preceding figure.

- `'looptransfer'` — Loop transfer. Specifies an output point before a loop opening followed by an input.

  Use this input/output type to compute the open-loop transfer function around the loop.

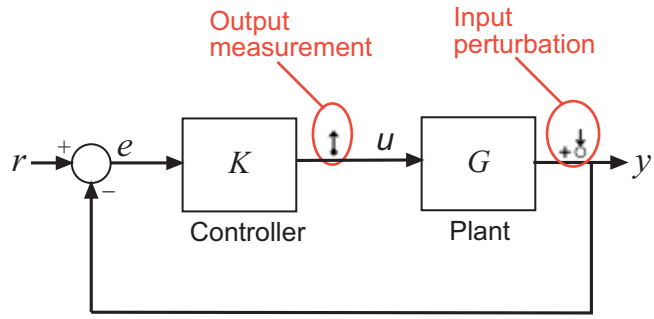
  For example, to compute `-GK` in the following feedback loop, specify the linearization input/output point as shown:



Similarly, compute `-KG` by specifying `looptransfer` at the output signal of the Controller block.

- `'input'` — Input perturbation. Specifies an additive input to a signal.

  For example, to compute the response `-K/(1+KG)` of the following feedback loop, specify an input perturbation and an output measurement point as shown:



  Similarly, you can compute `G/(1+GK)` using `input` at the output signal of the Controller block and an output measurement `output` at the output signal of the Plant block.

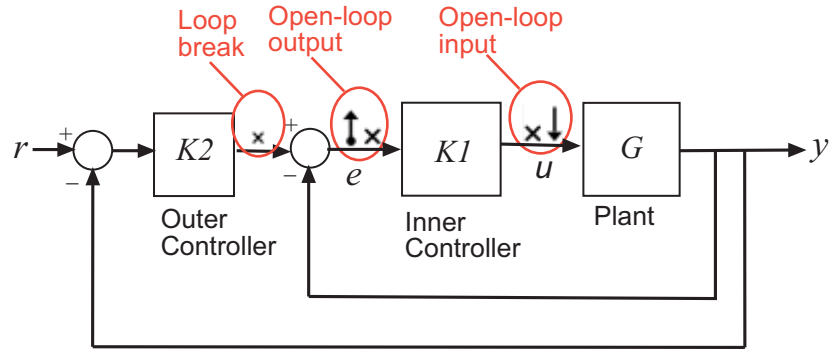
- `'output'` — Output measurement. Takes measurement at a signal.

  For example, to compute the response `-K/(1+KG)`, specify an output measurement point and an input perturbation as shown in the preceding figure.

- `'loopbreak'` — Loop break. Specifies a loop opening.

  Use to compute open-loop transfer function around a loop. Typically, you use this input/output type when you have nested loops or want to ignore the effect of some loops.

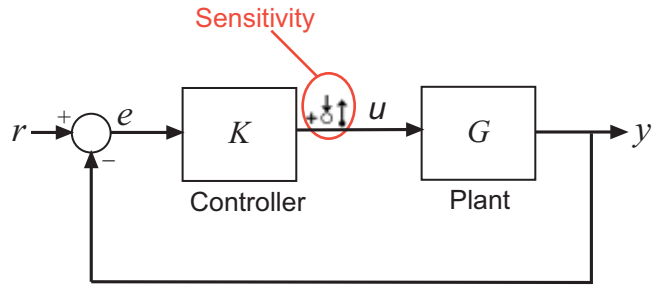
  For example, to compute the inner loop seen by `K1` and exclude the outer loop, specify the input/output points and `loopbreak` as shown:



- `'sensitivity'` — Sensitivity. Specifies an additive input followed by an output measurement.

  Use to compute sensitivity transfer function for an additive disturbance at the signal.

  For example, compute the input/load sensitivity, `1/(1+KG)`, in the following feedback loop, specify the linearization input/output point as shown:

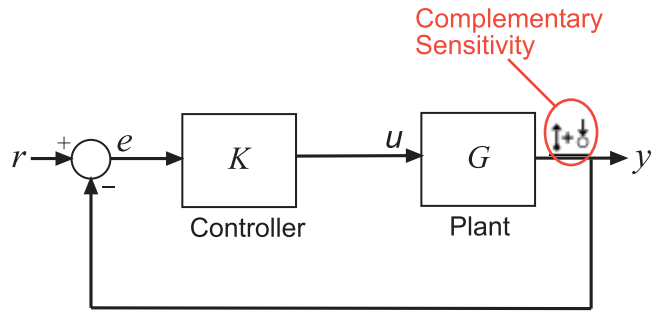


  Similarly, compute output sensitivity at the plant output, `1/(1+GK)`, by specifying a `sensitivity` input/output point at the output signal of the Plant block.

- `'compsensitivity'` — Complementary sensitivity. Specifies an output followed by an additive input.

  Use to compute closed-loop transfer function around the loop.

For example, to compute `-GK/(1+GK)` (the transfer function from `r` to `y`) in the following feedback loop, specify the linearization input/output point at the output signal of the Plant block as shown:



`io = linio('blockname', portnum, type, [], 'buselementname')` creates a linearization I/O object for the element `buselementname` at the bus signal that originates from the `portnum` of `blockname`.

Examples

This example shows how to create linearization I/O settings for a Simulink model.

1 Create an I/O setting for the signal originating from the Controller block of the `magball` model.

```
io(1)=linio('magball/Controller',1)
```

By default, this I/O is an input point that specifies an additive input to the signal.

```
1x1 vector of Linearization IOs:
-------------------------
1. Linearization input perturbation located at the following signal:
- Block: magball/Controller
- Port: 1
```

2 Create a second I/O setting within the object, `io`.

```
io(2)=linio('magball/Magnetic Ball Plant',1,'openoutput')
```

This I/O originates from the Magnetic Ball Plant block, is an output point and is also an open-loop point.

```
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
```

Select Individual Bus Element as Linearization I/O point

This example shows how to create a linearization I/O setting for individual bus elements in a bus signal.

1 Open Simulink model.

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

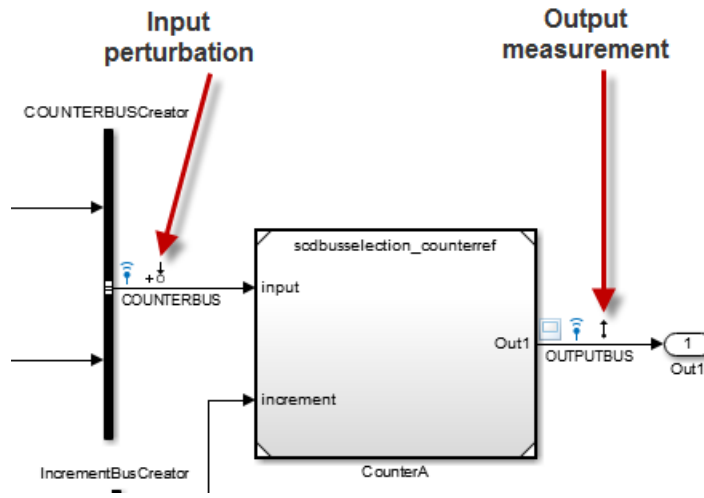
2 Specify to linearize the Counter block using linearization I/O points on an individual bus element.

```
io(1) = linio('scdbusselection/COUNTERBUSCreator',1,'input',[],...
      'limits.upper_saturation_limit');
io(2) = linio('scdbusselection/CounterA',1,'output',[],...
   'limits.upper_saturation_limit');
```

3 Update the model to reflect the linearization I/O object.

```
setlinio(mdl,io)
set_param(mdl,'ShowLinearizationAnnotations','on');
```

The linearization I/O markers appear in the model. Use these markers to visualize your linearization points.



4 Linearize the model at the model operating point.

```
sys = linearize(mdl,io);
```

See Also

`getlinio` | `linearize` | `setlinio`

Tutorials

- “Select Bus Elements as Linear Analysis Points” on page 2-27

linlft

Purpose

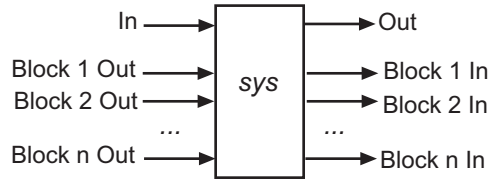
Linearize model while removing contribution of specified blocks

Syntax

```
lin_fixed = linlft(sys,io,blocks)
[lin_fixed,lin_blocks] = linlft(sys,io,blocks)
```

Description

`lin_fixed = linlft(sys,io,blocks)` linearizes the Simulink model named `sys` while removing the contribution of certain blocks. Specify the full block pathnames of the blocks to ignore in the cell array of strings 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`.

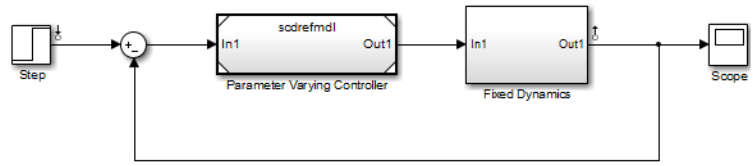
You can linearize the ignored blocks separately using `linearize`, and then combine the linearization results using `linlftfold`.

`[lin_fixed,lin_blocks] = linlft(sys,io,blocks)` returns the linearizations for each of the blocks specified in `blocks`. If `blocks` is a string identifying a single block path, `lin_blocks` is a single state-space (`ss`) model. If `blocks` is a cell 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.

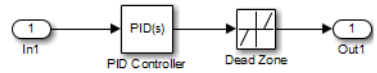
Examples

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

`linlftfold` | `linearize` | `linio` | `getlinio` | `operpoint` | `linearizeOptions`

linlftfold

Purpose

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

  You compute `lin_fixed` using `linlft`.

- Block linearizations for the blocks excluded from `lin_fixed`

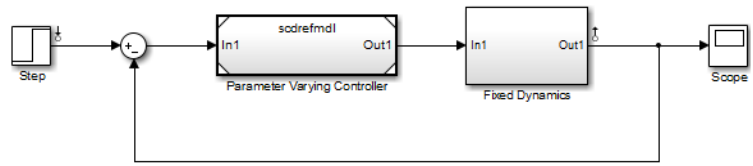
  You specify the block linearizations in a structure array `blocksubs`, which contains two fields:

  - `'Block'` is a string specifying the Simulink block to replace.
  - `'Value'` is the value of the linearization for each block.

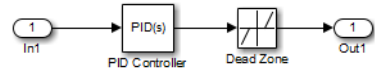
Examples

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

linlft | linearize | linio | getlinio | operpoint

linoptions

Purpose

Set options for linearization and finding operating points

Note `linoptions` will be removed in a future version. Instead, use:

- `linearizeOptions` — Create options for commands that perform linearization, such as `linearize`, `slTunable`, and `linlft`.
- `findopOptions` — Create options for operating point searches using `findop`.

Syntax

```
opt=linoptions
opt=linoptions('Property1','Value1','Property2','Value2',
   ...)
```

Alternatives

As an alternative to the `linoptions` function, set options for linearization and finding operating points in the Simulink Control Design GUI.

Description

`opt=linoptions` creates a linearization options object with the default settings. The variable, `opt`, is passed to the functions `findop` and `linearize` to specify options for finding operating points and linearization.

`opt=linoptions('Property1','Value1','Property2','Value2', ...)` creates a linearization options object, `opt`, in which the option given by `Property1` is set to the value given in `Value1`, the option given by `Property2` is set to the value given in `Value2`, etc. The variable, `opt`, is passed to the functions `findop` and `linearize` to specify options for finding operating points and linearization.

The following options can be set with `linoptions`:

`LinearizationAlgorithm`	Set to `'numericalpert'` to enable numerical-perturbation linearization (as in Simulink 3.0 software) where root-level inports and states are numerically perturbed. Linearization annotations are ignored and root-level inports and outports are used instead. Default is `'blockbyblock'`.
`SampleTime`	The time at which the signal is sampled. Nonzero for discrete systems, `0` for continuous systems, `-1` (default) to use the longest sample time that contributes to the linearized model.
`UseFullBlockNameLabels`	Set to `'off'` (default) to use truncated names for the linearization I/Os and states in the linearized model. Set to `'on'` to use the full block path to name the linearization I/Os and states in the linearized models.

`UseBusSignalLabels`	Set to `'off'` (default) to use bus signal channel number to label I/Os on bus signals in your linearization results. Set to `'on'` to use bus signal names to label I/Os on bus signals in your linearization results. 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 one of the following:
	  - Output of a bus creator block
	  - Root-level inport by passing through only virtual or nonvirtual subsystem boundaries

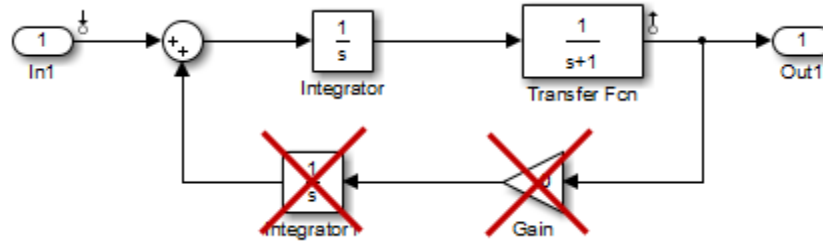
Note You cannot use this option when your model has mux/bus mixtures. For information on how to avoid buses used as muxes, see "Prevent Bus and Mux Mixtures" in the Simulink documentation.

`BlockReduction`

Set to `'on'` (default) to eliminate from the linearized model those blocks that are not in the path of the linearization. Block reduction eliminates the states of blocks in dead linearization paths from your linearization results. Some examples of dead linearization paths are linearization paths that include:

- Blocks that linearize to zero
- Switch blocks that are not active along the path
- Disabled subsystems
- Signals marked as open-loop linearization points

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).



Set to `'off'` to return a linearized model that includes all of the states of the model.

`IgnoreDiscreteStates` — Set to `'on'` when performing continuous linearization (SampleTime set to 0) to remove any discrete states from the linearization and accept the D value for all blocks with discrete states. Set to `'off'` (default) to include discrete states.

`RateConversionMethod` — When you linearize a multirate system, set this option to one of the following rate conversion methods:

- `'zoh'` (default) to use the zero order rate conversion method
- `'tustin'` to use the Tustin (bilinear) method
- `'prewarp'` to use the Tustin approximation with prewarping
- `'upsampling_zoh'` to upsample discrete states when possible and to use `'zoh'` otherwise
- `'upsampling_tustin'` to upsample discrete states when possible and to use `'tustin'` otherwise
- `'upsampling_prewarp'` to upsample discrete states when possible and to use `'prewarp'` otherwise

Note When you select `'prewarp'` or `'upsampling_prewarp'`, set the `PreWarpFreq` option to the desired prewarp frequency.

Note You can only upsample when you convert discrete states to a new sample time that is an *integer-value-times faster* than the sampling time of the original system.

	For more information, and examples, on methods and algorithms for rate conversions and linearization of multirate models, see: • Linearization of Multirate Models • Linearization Using Different Rate Conversion Methods • "Continuous-Discrete Conversion Methods" in the Control System Toolbox documentation
`PreWarpFreq`	The critical frequency Wc (in rad/sec) used by the `'prewarp'` option when linearizing a multirate system.
`UseExactDelayModel`	Set to `'on'` to return a linear model with an exact delay representation. Set to `'off'` (default) to return a model with approximate delays.
`NumericalPertRel`	Set the perturbation level for obtaining the linear model (default value is `1e-5`). The perturbation of the system's states is specified by: $\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times \lvert x \rvert$ The perturbation of the system's inputs is specified by: $\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times \lvert u \rvert$
`NumericalXPert`	Individually set the perturbation levels for the system's states using an operating point object. Use the `operpoint` function to create an operating point object for the model.
`NumericalUPert`	Individually set the perturbation levels for the system's inputs using an operating point object. Use the `operpoint` function to create an operating point object for the model.
`OptimizationOptions`	Set options for use with the optimization algorithms. These options are the same as those set with `optimset`. For more information on these algorithms, see the Optimization Toolbox documentation.

`OptimizerType`

Set optimizer type to be used by trim optimization if the Optimization Toolbox software is installed. The available optimizer types are:

- `graddescent_elim`, the default optimizer, enforces 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' value. The optimizer fixes the states, `x`, and inputs, `u`, that are marked as 'Known' in an operating point specification and then optimizes the remaining variables.
- `graddescent`, enforces 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' value. `findop` also minimizes the error between the states, `x`, and inputs, `u`, that are marked as 'Known' in 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.
- `lsqnonlin` fixes the states, `x`, and inputs, `u`, that are marked as 'Known' in an operating point specification and optimizes the remaining variables. The algorithm then 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' value.
- `simplex` uses the same cost function as `lsqnonlin` with the direct search optimization routine found in `fminsearch`.

	See the Optimization Toolbox documentation for more information on these algorithms. If you do not have the Optimization Toolbox software, you can access the documentation at `http://www.mathworks.com/support/`.
`DisplayReport`	Set to `'on'` to display the operating point summary report when running `findop`. Set to `'off'` to suppress the display of this report.

See Also

`findop` | `linearize`

Purpose

Create operating point for Simulink model

Syntax

```
op = operpoint('sys')
```

Alternatives

As an alternative to the `operpoint` function, create operating points in the Linear Analysis Tool. See "Steady-State Operating Points (Trimming) from Specifications" on page 1-14.

Description

`op = operpoint('sys')` returns an object, `op`, containing the operating point of a Simulink model, `sys`. Use the object with the function `linearize` to create linearized models. The operating point object properties are:

- "Model" on page 7-145
- "States" on page 7-145
- "Inputs" on page 7-146
- "Time" on page 7-146

Edit the properties of this object directly or with the `set` function.

Model

`Model` specifies the name of the Simulink model that this operating point object refers to.

States

`States` describes the operating points of states in the Simulink model. The `States` property is a vector of state objects that contains the operating point values of the states. There is one state object per block that has a state in the Simulink model. The `States` object has the following properties:

`Nx`	Number of states in the block. This property is read-only.
`Block`	Block with which the states are associated.

`x`	Vector containing the values of states in the block.
`Ts`	Vector containing the sample time and offset for the state.
`SampleType`	Set this value to `CSTATE`, for a continuous state, or `DSTATE` for a discrete state.
`inReferencedModel`	Set this value to `1`, when the state is inside a referenced model, or `0`, when it is not.
`Description`	Text string describing the block.

Inputs

`Inputs` is a vector of input objects that contains the input levels at the operating point. There is one input object per root-level inport block in the Simulink model. The `Inputs` object has the following properties:

`Block`	Inport block with which the input vector is associated
`PortWidth`	Width of the corresponding inport
`u`	Vector containing the input level at the operating point
`Description`	Text string describing the input

Time

`Time` specifies the time at which any time-varying functions in the model are evaluated.

Examples

To create an operating point object for the Simulink model `magball`, type:

```
op = operpoint('magball')
```

which returns the following:

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

MATLAB software displays the name of the model, the time at which any time-varying functions in the model are evaluated, the names of blocks containing states, and the values of the states at the operating point. In this example there are four blocks that contain states in the model and four entries in the `States` object. The first entry contains two states. MATLAB also displays the `Inputs` although there are not any in this model. To view the properties of `op` in more detail, use the `get` function.

See Also

`get` | `linearize` | `operspec` | `set` | `update`

operspec

Purpose

Operating point specifications

Syntax

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

Description

`opspec = operspec(sys)` returns the operating point specifications object for steady state operating point analysis using `findop`. The Simulink model must be open.

Tips

- Use `get` to display the operating point specification object properties.

Input Arguments

sys

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

Output Arguments

opspec

Operating point specification object.

After creating the operating point object, you can modify the operating point states and input levels. For example, `opspec.States(1).Known = 1` specifies that the first model state value is known, and the value of the known state `opspec.States(1).x = 2`.

The operating point object has these properties:

- `Model` — Simulink model name. String.
- `States` — State operating point specification. Vector of data structures, where each data structure represents the states of one Simulink block. Each `States` structure has these fields:

`x`	Simulink block state values, specified as a vector of states. Also set the `Known` field of the `States` property for known state values that remain fixed during operating point search.
`Known`	Known state value specification: • `1` — Known value that is fixed during operating point search. ▪ `0` (default) — Unknown value to be found by optimization. Also specify the known operating point values using the `x` field of the `States` structure.
`SteadyState`	Steady state value specification: • `1` (default) — Equilibrium state. ▪ `0` — Nonequilibrium state.
`Min`	Minimum bounds on the state value, specified as a scalar or vector.
`Max`	Maximum bounds on the state value, specified as a scalar or vector.
`Ts`	(Only for discrete-time states) Sample time and offset of each Simulink block state, specified as a vector.
`Description`	Block state description, specified as a string.
`Nx`(read only)	Number of states in the Simulink block.
`Block`	Simulink block name.

`SampleType` State time rate can have the 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.

- `Inputs` — Input level specifications at the operating point. Vector of input specification objects, where each object represents the input levels of one root-level inport block in the Simulink block. Each input specification object has these properties:

  `u` Inport block input levels at the operating point, specified as a vector of input levels.

  Also set the `Known` field of the `Inputs` property for known input levels that remain fixed during operating point search.

  `Known` Known input level specification:

  - `1` — Known input level that is fixed during operating point search.
  - `0` (default) — Unknown input level to be found by optimization.

  Also specify the known operating point input levels using the `u` property of the input specification object.

  `Min` Minimum bounds on the input level, specified as a scalar or vector.

`Max`	Maximum bounds on the input level, specified as a scalar or vector.
`Description`	Inport block input description, specified as a string.
`Block`	Inport block name.
`PortWidth`	Number of inport block signals.

- `Outputs` — Output level specifications at the operating point. Vector of output specification objects, where each object represents one output specification per root-level output block in the Simulink block. You can constrain additional output levels using `addoutputspec` to add another output specification.

  Each output specification object has these properties:

`y`	Outport block output levels at the operating point, specified as a vector of output levels. Also set the `Known` field of the `Outputs` property for known output levels that remain fixed during operating point search.
`Known`	Known output level specification: • `1` — Known output level constraint that must be met during operating point search. ▪ `0` (default) — Unknown input level to be found by optimization. Also specify the known operating point output levels using the `y` property of the output specification object.

`Min`	Minimum bounds on the output level, specified as a scalar or vector.
`Max`	Maximum bounds on the output level, specified as a scalar or vector.
`Description`	Outport block input description, specified as a string.
`Block`	Outport block name.
`PortWidth`	Number of outport block signals.

- `Time` — Time instants for evaluating the time-varying functions in the model.

Examples

Steady-State Operating Point (Trimming) From Specifications

This example shows how to use `findop` to compute an operating point of a model from specifications.

1 Open Simulink model.

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

2 Create operating point specification object.

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

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

```
 Operating 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
----------
```

`operspec` extracts the default operating point of the Simulink model with two states. The model does not have any root-level inport blocks and no root-level outport blocks or output constraints.

3 Configure specifications for the first model state.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

The first state must be at steady state and have an initial value of 2 with a lower bound of 0.

4 Configure specifications for the second model state.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

The second state sets the desired height of the water in the tank at 10. Configuring the height as a known value keeps this value fixed when computing the operating point.

5 Find the operating point that meets these specifications.

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

`opreport` describes how closely the optimization algorithm met the specifications at the end of the operating point search.

```
 Operating 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` indicates the time derivative of each state. The actual `dx` values of zero indicate that the operating point is at steady state. The desired `dx` value is in parentheses.

Initialize Steady-State Operating Point Search Using Simulation

This example shows how to use `findop` to compute an operating point of a model from specifications, where the initial state values are extracted from a simulation snapshot.

1. Open Simulink model.

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

2. Extract an operating point from simulation after 10 time units.

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

3. Create operating point specification object.

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

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

4 Configure initial values for operating point search.

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

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

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

`opreport` describes the optimization algorithm status at the end of the operating point search.

```
 Operating 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:     -1.1e-014 (0)

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

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

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

Operating Point (Trim Analysis) With Output Constraint

This example shows how to use `addoutputspec` to specify an output constraint to the operating point specification object for computing the operating point.

1 Open Simulink model.

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

2 Create operating point specification object.

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

By default, `opspec` specifies that the operating point is at steady state, or equilibrium.

```
 Operating 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
----------
```

`operspec` extracts the default operating point of the Simulink model with two states and one root-level inport block. There are no root-level outport blocks or output constraints.

3 Fix the first output port of the Vehicle Dynamics to 2000 RPM.

```
opspec = addoutputspec(op_spec,'scdspeed/rad//s to rpm',1);
opspec.Outputs.Known = 1;
opspec.Outputs.y = 2000;
```

4 Find the operating point that meets this specification.

```
op = findop(sys,op_spec)

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

 Operating 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-013 (0)
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      x:           209      dx:    -8.48e-012 (0)

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

Outputs:
----------
(1.) scdspeed/rad//s to rpm
      y:        2e+003    (2e+003)
```

See Also

addoutputspec | findop | update

Purpose

Set properties of linearization I/Os and operating points

Syntax

```
set(ob)
set(ob,'PropertyName',val)
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

Purpose

Specify linearization input/output (I/O) points for Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
oldio = setlinio('sys',io)
oldio = setlinio('blockpath',io)
```

Alternatives

As an alternative to the `setlinio` function, edit linearization I/Os annotated in the Simulink model using the:

- **Exact Linearization** tab of the Linear Analysis Tool. In the **Setup** section, click to view and edit the linearization I/Os. The icon appears only when **Analysis I/Os** is set to `Model I/Os`.
- **Linearization inputs/outputs** table and **Click a signal in the model to select it** in the **Linearizations** tab of the Block Parameters dialog box for Linear Analysis Plots or Model Verification blocks.

Description

`oldio = setlinio('sys',io)` assigns the settings in the vector of linearization input/output (I/O) objects, `io`, to the Simulink model, `sys`. These settings appear as annotations on the signal lines. `oldio` contains the old I/O settings. Use the function `getlinio` or `linio` to create the linearization I/O objects. You can save I/O objects to disk in a MAT-file and use them later to restore linearization settings in a model.

`oldio = setlinio('blockpath',io)` assigns the settings in `io` as the linearization I/Os in a Linear Analysis Plots block or a Model Verification block. `blockpath` is the full path to the block.

Examples

This example shows how to assign linearization input/output settings to a Simulink model.

Before assigning I/O settings to a Simulink model using `setlinio`, you must create a vector of I/O objects representing linearization annotations, such as input points or output points, on a Simulink model.

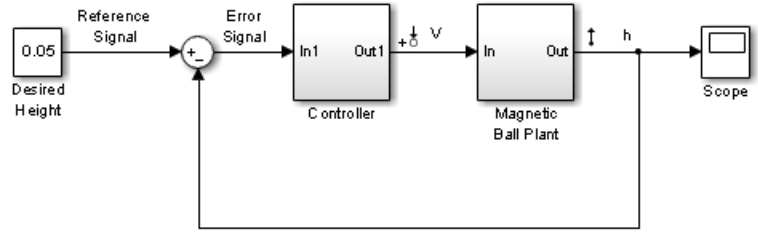
1 Open a Simulink model.

```
magball
```

2 Right-click the signal line between the Magnetic Ball Plant and the Controller. Select **Linear Analysis Points > Input Perturbation** to place an input point on this signal line. A small arrow pointing to a small circle just above the signal line represents the input point. The input point is not the output of the block, rather it is an additive input to the signal.

3 Right-click the signal line after the Magnetic Ball Plant. Select **Linear Analysis Points > Output Measurement** from the menu to place an output point on this signal line.

The model diagram should now look similar to the following figure:



4 Create an I/O object with the `getlinio` function:

```
io=getlinio('magball')
```

5 Modify `io` to compute the plant transfer function. Edit the object or use the `set` function.

```
io(2).Type='openoutput';
```

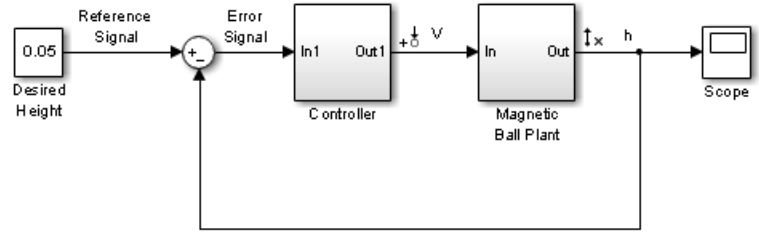
6 Assign the new settings in `io` to the model.

```
oldio=setlinio('magball',io)
```

This assignment returns the old I/O settings (that have been replaced by the settings in `io`).

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal
- Block: magball/Controller
- Port: 1

2. Linearization output measurement located at the following signal
- Block: magball/Magnetic Ball Plant
- Port: 1
```

The model diagram now looks similar to the following figure.

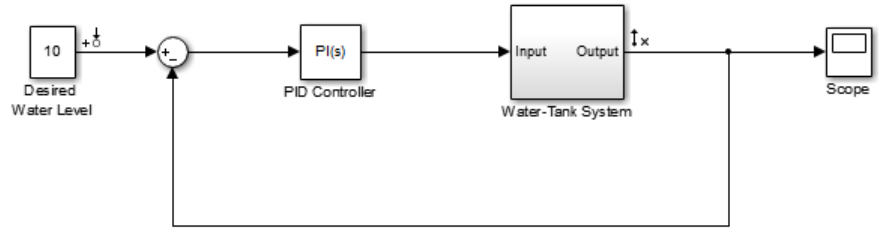


Update linearization input/output settings in a Linear Analysis Plots block

This example shows how to update linearization input/output settings in a Linear Analysis Plots block.

1. Open the `watertank` model, and specify input and output (I/O).
   a. Right-click the Desired Water Level output signal, and select **Linear Analysis Points > Input Perturbation**.
   b. Right-click the Water-Tank System output signal, and select **Linear Analysis Points > Output Measurement**.

The linearization I/O markers appear in the model, as shown in the next figure.



Alternatively, you can use `linio`.

2 Drag and drop a Bode Plot block from the Simulink Control Design Linear Analysis Plots library into the Simulink Editor. When you drag and drop the block, the block I/Os are set to the model I/Os.

3 Find all I/Os used by the Bode Plot block.

```
io = getlinio('watertank/Bode Plot')
```

The following results appear at the MATLAB prompt:

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: watertank/Desired  Water Level
- Port: 1

2. Linearization output measurement located at the following signal:
- Block: watertank/Water-Tank System
- Port: 1
```

4 Specify the linearization output to be open loop.

```
io(2).Type = 'openoutput';
```

Note The loop opening does not affect the model I/Os.

5 Update the I/O in the Bode Plot block.

```
oldio = setlinio('watertank/Bode Plot',io);
```

See Also

`get` | `getlinio` | `linio` | `set`

Purpose

Set states and inputs in operating points

Syntax

```
op_new=setxu(op_point,x,u)
```

Alternatives

As an alternative to the `setxu` function, set states and inputs of operating points with the Simulink Control Design GUI.

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**. In the **Save to workspace** pane, select **Final states**. 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
```

See Also

getxu | initopspec | operpoint | operspec

Purpose

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,opt)
sllin = slLinearizer(mdl,pt,op,param,blocksub,opt)
```

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 of `sllin`. The interface additionally adds the linear analysis points that imply an opening as permanent openings.

`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,opt)` configures the linearization algorithm options.

`sllin = slLinearizer(mdl,pt,op,param,blocksub,opt)` uses any combination of the input arguments `pt`, `op`, `param`, `blocksub`, and `opt` 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 additionally 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 and permanent openings 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.

Commands that extract linearizations from the `slLinearizer` interface recompile the Simulink model if you changed any interface properties since the last linearization. The commands also recompile the Simulink model if you made calls to specific functions since the last linearization. These functions include `addPoint`, `addOpening`, `removePoint`, `removeOpening`, `removeAllPoints`, and `removeAllOpenings`.

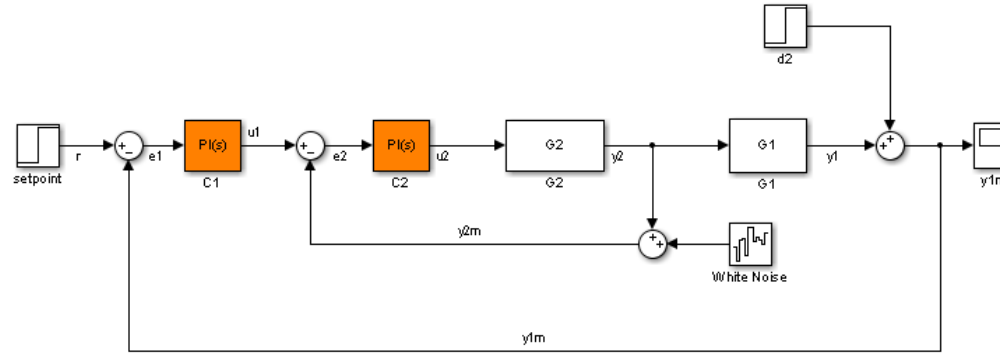
Examples

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 - Name of Simulink model to be linearized

string

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

Example: `'scdcascade'`

pt - Analysis point

string | cell array of strings | vector of linearization I/O objects

Analysis point to be added to the list of analysis points for `sllin`, specified as:

- String — Analysis point identifier that can be any of the following:

- Analysis point signal name, for example `pt = 'torque'`
- Block path for a block with a single output port, for example `pt = 'Motor/PID'`
- Path to block and port originating the analysis point, for example `pt = 'Engine Model/1'` or `pt = 'Engine Model/torque'`

- Cell array of strings — Specifies multiple analysis point identifiers. For example:

```
pt = {'torque','Motor/PID'}
```

- Vector of linearization I/O objects — Use `linio` to create `pt`. 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 additionally adds all `'loopbreak'` type signals as permanent openings.

param - Parameter samples for linearizing `mdl`

structure | structure array

Parameter samples for linearizing `mdl`, specified as:

- Structure — For a single parameter, `param` must be a structure with the following fields:
  - `Name` — Parameter name, specified as a string or MATLAB expression
  - `Value` — Parameter sample values, specified as a double array

  For example:

```
param.Name = 'A';
param.Value = linspace(0.9*A,1.1*A,3);
```

- Structure array — Vary the value of multiple parameters. For example, suppose you want to vary the value of the `A` and `b` model parameters 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 additionally configure `sllin.OperatingPoints` with operating point objects only, the software uses single model compilation.

op - Operating point for linearizing `mdl`

operating point object | array of operating point objects | array of positive scalars

Operating point for linearizing `mdl`, specified as:

- Operating point object, created using `findop`.

  For example:

```
op = findop('magball',operspec('magball'));
```

- Array of operating point objects — Specifies multiple operating points.

  For example:

```
op = findop('magball',[10 20]);
```

- Array of positive scalars, specifying simulation snapshot times.

  For example:

```
op = [1 4.2];
```

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 `sllin.Parameters`. When you batch linearize `mdl`, the software uses only one model compilation.
- 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. Use `blocksub` to specify a custom linearization for a block or subsystem. You also can use `blocksub` for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

`blocksub` is an *n*-by-1 structure, where *n* is the number of blocks for which you specify the linearization. `blocksub` has these fields:

- `Name` — Block path corresponding to the block for which you want to specify the linearization.

  `blocksub.Name` is a string of the form *model/subsystem/block* that uniquely identifies a block in the model.

- `Value` — Desired linearization of the block, specified as a structure with these fields:

  `Specification` — Block linearization, specified as a string. The string can include a MATLAB expression or function that returns one of the following:

  - Linear model in the form of a D-matrix
  - Control System Toolbox LTI model object

- Robust Control Toolbox uncertain state space or uncertain real object (requires Robust Control Toolbox software)

If `blocksub.Value.Specification` is a MATLAB expression, this expression must follow the resolution rules , as described in "Symbol Resolution".

If `blocksub.Value.Specification` is a function, this function must have one input argument, `BlockData`, which is a structure that the software creates automatically and passes to the specification function. `BlockData` has the following fields:

- `BlockName` is the name of the Simulink block with the specified linearization.
- `Parameters` is a structure array containing the evaluated values for the block. Each element of the array has the fields `'Name'` and `'Value'`, which contain the name and evaluated value, respectively, for the parameter.
- `Inputs` is a structure that has the following fields:
  - `BlockName` — Contains the name of the block whose output connects to the input of the block whose linearization you are specifying. For example, if you specify the linearization of a block called `Dynamics`, and the second input of `Dynamics` is driven by a signal from a block called `Torque`, then `BlockData.Inputs(2).BlockName` is the full block path name of `Torque`.
  - `PortIndex` — Identifies which output port of `BlockName` corresponds to the input of the block whose linearization you are specifying. For example, if the third output from `Torque` drives the second input of `Dynamics`, then `BlockData.Inputs(2).PortIndex = 3`.
  - `Values` — The value of the signal specified by `BlockName` and `PortIndex`. If this signal is a vector-valued signal, `Values` is a vector of corresponding dimension.
- `ny` is the number of output channels of the block linearization.

- `nu` is the number of input channels of the block linearization.
- `Type` — Specification type, specified as one of these strings:

  ```
  'Expression'
  'Function'
  ```

- `ParameterNames` — Linearization function parameter names, specified as a comma-separated list of strings. Specify only when `blocksub.Value.Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block

  You also must specify the corresponding `blocksub.Value.ParameterValues` field.

- `ParameterValues` — Linearization function parameter values that correspond to `blocksub.Values.ParameterNames`. Specify only when `blocksub.Value.Type = 'Function'`.

  `blocksub.Value.ParameterValues` is a comma separated list of values. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`.

- `BlockLinearization` is a state-space (`ss`) model that is the current default linearization of the block. You can use `BlockData.BlockLinearization` in the specification function to specify a block linearization that depends on the default linearization, such as the default linearization multiplied by a time delay.

opt - Linearization algorithm options

options set created using `linearizeOptions`

Linearization algorithm options, specified as an options set created using `linearizeOptions`.

Example: `opt = linearizeOptions('LinearizationAlgorithm','numericalpert')`

Properties

`slLinearizer` objects 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 — For a single parameter, `param` must be a structure with the following fields:
  - `Name` — Parameter name, specified as a string or MATLAB expression
  - `Value` — Parameter sample values, specified as a double array

  For example:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, suppose you want to vary the value of the `A` and `b` model parameters 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 additionally 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`.

  For example:

  ```
  op = findop('magball',operspec('magball'));
  ```

- Array of operating point objects — Specifies multiple operating points.

  For example:

  ```
  op = findop('magball',[10 20]);
  ```

- Array of positive scalars, specifying simulation snapshot times.

  For example:

  ```
  op = [1 4.2];
  ```

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 `sllin.Parameters`. When you batch linearize `mdl`, the software uses only one model compilation.
- 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 options set created using `linearizeOptions`.

Set this property using the `opt` input argument or dot notation (`sllin.Options = opt`).

Model

Name of the Simulink model to be linearized, specified as a string by the input argument `mdl`.

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

Algorithms

`slLinearizer` linearizes your Simulink model using the algorithms described in "Exact Linearization Algorithm" on page 2-152.

See Also

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

Related Examples

- "Batch Compute Steady-State Operating Points" on page 1-38
- "Specify Parameter Samples" on page 2-190
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169

Concepts

- "What Is Batch Linearization?" on page 2-160
- "How the Software Treats Loop Openings" on page 2-158
- "Batch Linearization Efficiency When You Vary Parameter Values" on page 2-164

Purpose

Add signal to list of openings for `slLinearizer` interface

Syntax

```
addOpening(sllin,pt)

addOpening(sllin,blk,port_num)
addOpening(sllin,blk,port_num,bus_elem_name)
```

Description

`addOpening(sllin,pt)` adds the specified point (signal) to the list of permanent openings for the `slLinearizer` interface, `sllin`.

Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

`addOpening(sllin,blk,port_num)` adds the signal at the specified output port of the specified block as a permanent opening for the `slLinearizer` interface, `sllin`.

`addOpening(sllin,blk,port_num,bus_elem_name)` adds the specified bus element as a permanent opening.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Opening

string | cell array of strings | vector of linearization I/O objects

Opening to add to the list of permanent openings for `sllin`, specified as:

- 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 strings — 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 `sllin`.

blk - Block path identifying block where opening originates

string (default) | cell array of strings

Block path identifying the block where the opening originates, specified as a string or cell array of strings.

Dimensions of `blk`:

- For a single opening, specify `blk` as a string.

  For example, `blk = 'scdcascade/C1'`.

- For multiple openings, specify `blk` as a cell array of strings. `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

string (default) | cell array of strings

Bus element name, specified as a string or cell array of strings.

Dimensions of `bus_elem_name`:

- For a single opening, specify `bus_elem_name` as a string.

  For example, `bus_elem_name = 'upper_saturation_limit'`.

- For multiple openings, specify `bus_elem_name` as a cell array of strings. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `bus_elem_name = {'upper_saturation_limit','lower_saturation_limit'}`.

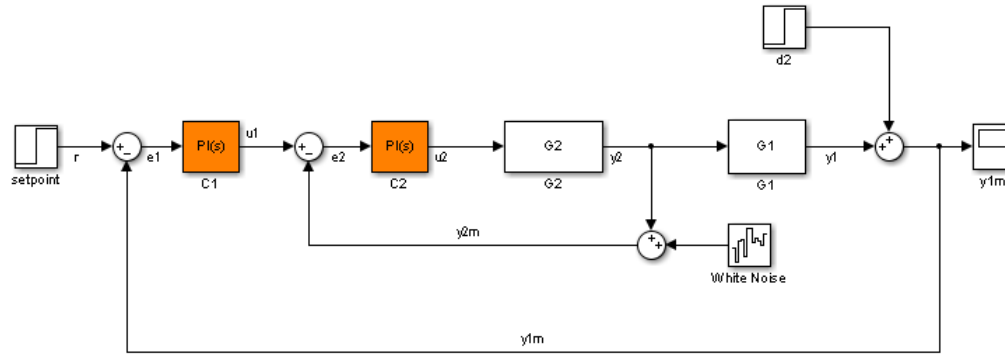
Examples

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('scdcascade');
```

Add the `y1m` signal as a permanent opening of `sllin`.

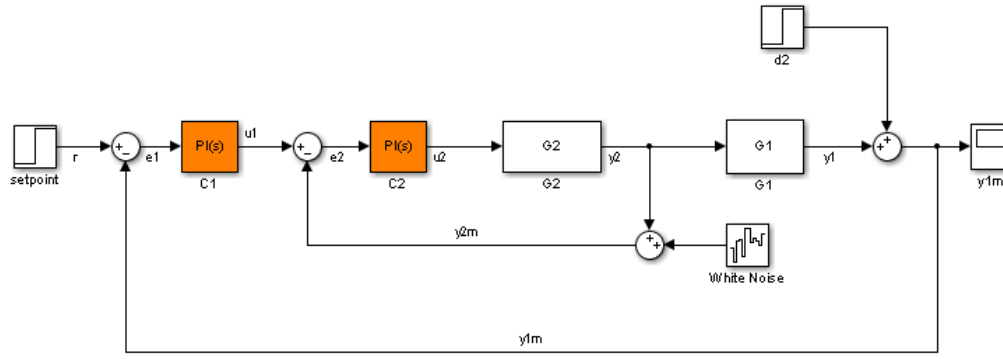
```
addOpening(sllin,'y1m');
```

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('scdcascade');
```

Add the `y1m` signal as a permanent opening of `sllin`.

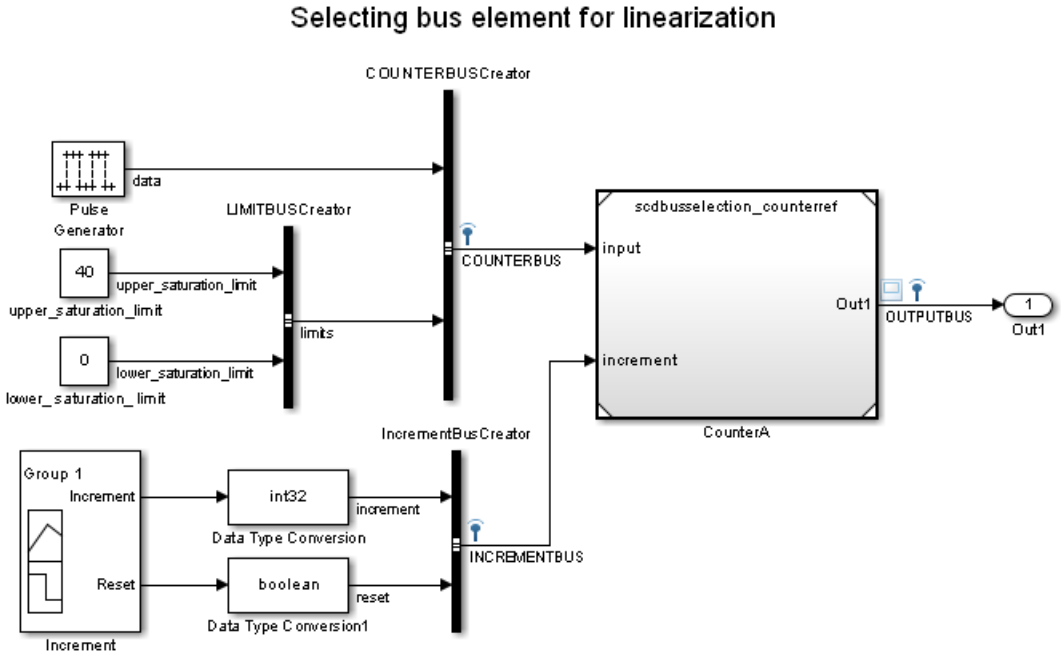
```
addOpening(sllin,'scdcascade/Sum',1)
```

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 `lower_saturation_limit` bus elements as openings to `sllin`.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUSCre
port_num = [1 1];
bus_elem_name = {'upper_saturation_limit','lower_saturation_limit'};

addOpening(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.

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addPoint | linio | removeOpening | removeAllOpenings`

addPoint

Purpose

Add signal to list of analysis points for `slLinearizer` interface

Syntax

```
addPoint(sllin,pt)

addPoint(sllin,blk,port_num)
addPoint(sllin,blk,port_num,bus_elem_name)
```

Description

`addPoint(sllin,pt)` adds the specified point to the list of analysis points for the `slLinearizer` interface, `sllin`.

You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

`addPoint(sllin,blk,port_num)` adds the point that originates at the specified output port of the specified block as an analysis point for the `slLinearizer` interface, `sllin`.

`addPoint(sllin,blk,port_num,bus_elem_name)` adds the specified bus element as an analysis point.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Analysis point

string | cell array of strings | vector of linearization I/O objects

Analysis point to add to the list of analysis points for `sllin`, specified as:

- 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'`

To specify multiple signal identifiers, specify `pt` as a cell array of strings.

- Cell array of strings — 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

string (default) | cell array of strings

Block path identifying the block where the analysis point originates, specified as a string or cell array of strings.

Dimensions of `blk`:

- For a single point, specify `blk` as a string.

  For example, `blk = 'scdcascade/C1'`.

- For multiple points, specify `blk` as a cell array of strings. `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 analysis point originates

positive integer (default) | vector of positive integers

Port where the analysis point originates, specified as a positive integer or a vector of positive integers.

Dimensions of `port_num`:

- For a single point, specify `port_num` as a positive integer.

  For example, `port_num = 1`.

- For multiple points, 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

string (default) | cell array of strings

Bus element name, specified as a string or cell array of strings.

Dimensions of `bus_elem_name`:

- For a single point, specify `bus_elem_name` as a string.

  For example, `bus_elem_name = 'upper_saturation_limit'`.

- For multiple points, specify `bus_elem_name` as a cell array of strings. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `bus_elem_name = {'upper_saturation_limit','lower_saturation_limit'}`.

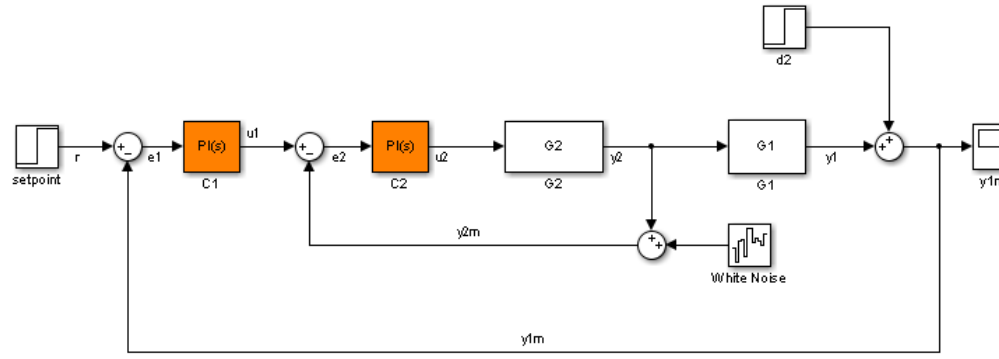
Examples

Add Analysis Point Using Signal Name

Add the `u1` and `y1` signals of the `scdcascade` model to an `slLinearizer` interface.

Open the `scdcascade` model.

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



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer('scdcascade');
```

Add `u1` and `y1` as analysis points for `sllin`.

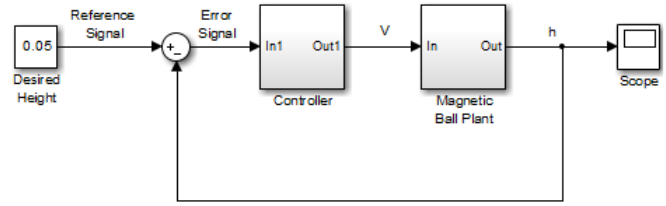
```
addPoint(sllin,{'u1','y1'});
```

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 Design Height and Magnetic Ball Plant blocks as analysis points to an `slLinearizer` interface.

Open the `magball` model.

```
open_system('magball');
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer('magball');
```

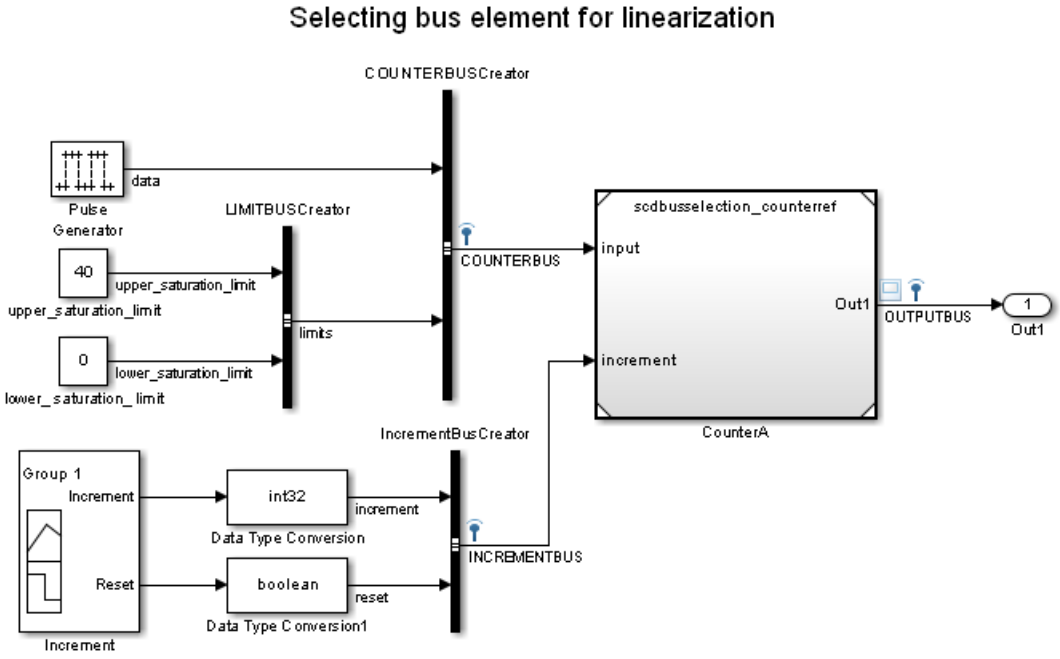
Add the signals originating at the Design Height and Magnetic Ball Plant blocks as analysis points of `sllin`. Both signals originate at the first (and only) port of the respective blocks.

```
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 for the model.

```
sllin = slLinearizer(mdl);
```

The `COUNTERBUS` signal of `scdbusselection` contains multiple bus elements. Add the `upper_saturation_limit` and `lower_saturation_limit` bus elements as analysis points to `sllin`.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUS
port_num = [1 1];
bus_elem_name = {'upper_saturation_limit','lower_saturation_limit'};

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.

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations.

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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addOpening | linio | removePoint | removeAllPoints`

getCompSensitivity

Purpose

Complementary sensitivity function at specified point using `slLinearizer` interface

Syntax

```
sys = getCompSensitivity(sllin,pt)
sys = getCompSensitivity(sllin,pt,temp_opening)
sys = getCompSensitivity( ___ ,mdl_index)
```

Description

`sys = getCompSensitivity(sllin,pt)` returns the complementary sensitivity function at the specified analysis point for the model associated with the `slLinearizer` interface, `sllin`.

The software enforces all the permanent openings specified for `sllin` when it calculates `sys`. If you configured either `sllin.Parameters`, or `sllin.OperatingPoints`, or both, `getCompSensitivity` performs multiple linearizations and returns an array of complementary sensitivity functions.

`sys = getCompSensitivity(sllin,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.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Analysis point signal name

string | cell array of strings

Analysis point signal name, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

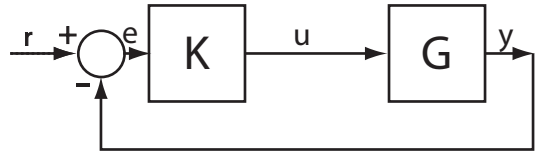
  You can specify `pt` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

  For example, `pt = 'y1m'`.

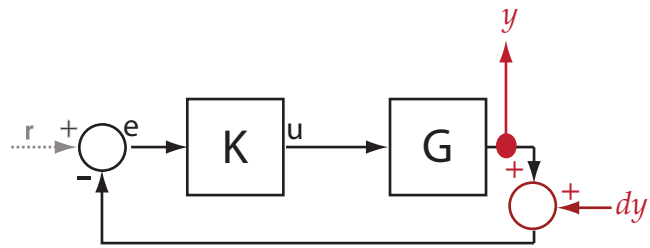
- Cell array of strings — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `sys`, the software adds a linearization output, followed by a linearization input at `pt`.

Consider the following model:

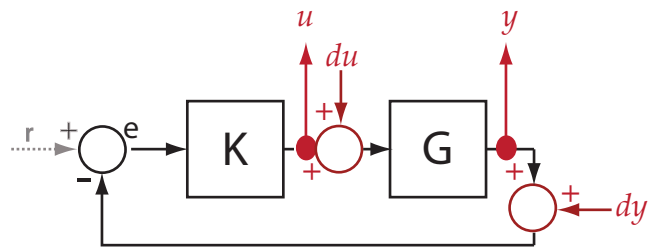


Specify `pt` as `'y'`:



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

string | cell array of strings

Temporary opening signal name, specified as:

- String — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `sllin`. To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 `temp_opening` as the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

You can specify `temp_opening` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

For example, `temp_opening = 'y1m'`.

- Cell array of strings — 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 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 = sllin.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 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 = sllin.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

`ss` model

Complementary sensitivity function, returned as described below:

- If you did not configure `sllin.Parameters` and `sllin.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 an `ss` model
- If you configured `sllin.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as an `ss` model array of the same size as `sllin.OperatingPoints`.
- If you configured `sllin.Parameters` and specified `sllin.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 an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `sllin.OperatingPoints` is the same size as the parameter grid specified by `sllin.Parameters`. The software computes each linearization using corresponding

operating points and parameter grid points. `sys` is returned as an `ss` model array of the same size as the parameter grid

- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as an `ss` model array of size `N`-by-`p`.

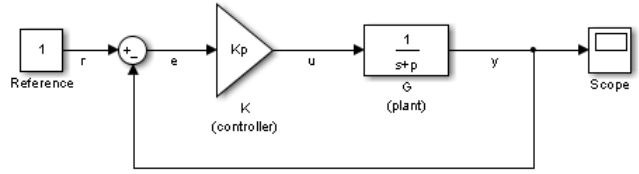
Examples

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`.

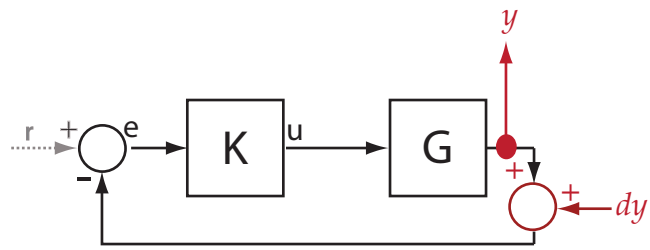
```
addPoint(sllin,'y');
```

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`.



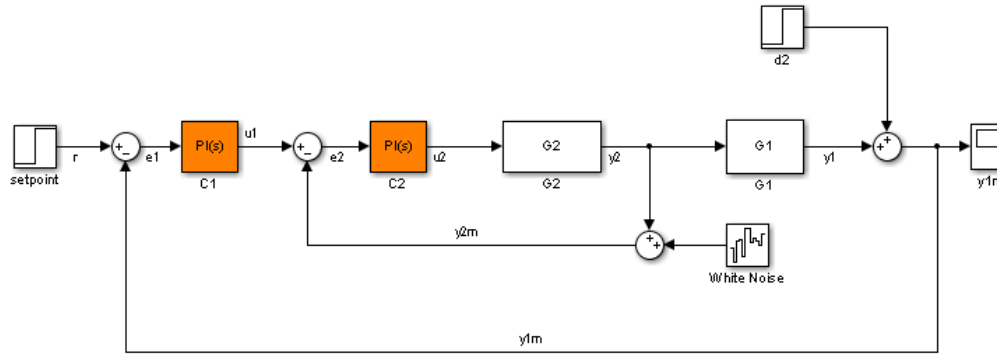
`sys` is the transfer function from `dy` to `y`, which is equal to $-(I+GK)^{-1}GK$.

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.

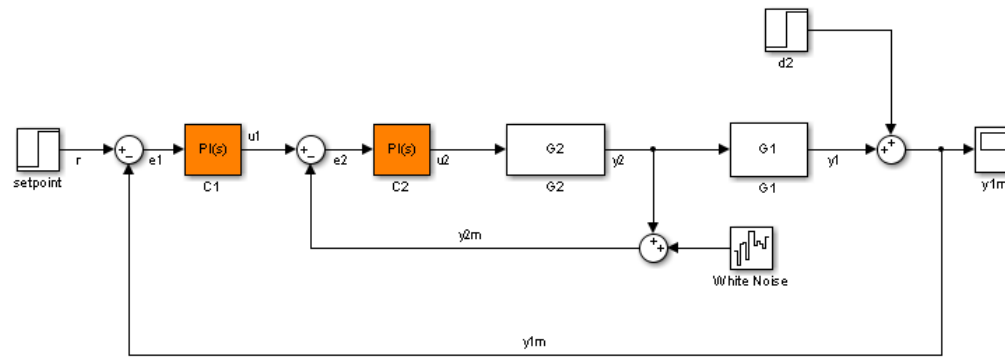
Complementary Sensitivity Function for Specific Parameter Combination

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
```

Obtain the complementary sensitivity transfer function at `y2`.

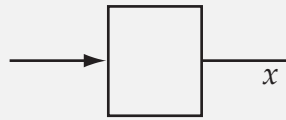
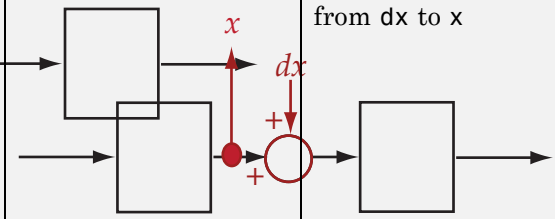
```
sys = getCompSensitivity(sllin,'y2','y1m',mdl_index);
```

Definitions

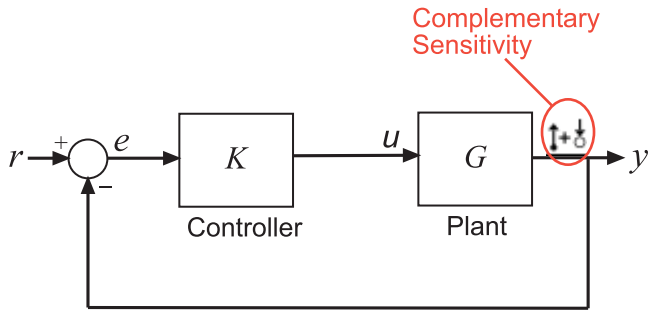
Complementary Sensitivity Function

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 the 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:

$$
\begin{aligned}
& y = -GK(y + dy) \\
& \rightarrow y = -GKy - GKdy \\
& \rightarrow (I + GK)y = -GKdy \\
& \rightarrow y = \underbrace{-(I + GK)^{-1} GK}_{T} dy.
\end{aligned}
$$

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` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Loop Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addPoint | addOpening | getIOTransfer | getLoopTransfer | getSensitivity`

Related Examples

- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177

- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "How the Software Treats Loop Openings" on page 2-158

getIOTransfer

Purpose

Transfer function for specified I/O set using `slLinearizer` interface

Syntax

```
sys = getIOTransfer(sllin,in,out)
sys = getIOTransfer(sllin,in,out,temp_opening)

sys = getIOTransfer(sllin,ios)

sys = getIOTransfer(___,mdl_index)
```

Description

`sys = getIOTransfer(sllin,in,out)` returns the transfer function for the specified inputs and outputs for the model associated with the `slLinearizer` interface, `sllin`.

The software enforces all the permanent openings specified for `sllin` when it calculates `sys`. For information on how `getIOTransfer` treats `in` and `out`, see "Transfer Functions" on page 7-222. If you configured either `sllin.Parameters`, or `sllin.OperatingPoints`, or both, `getIOTransfer` performs multiple linearizations and returns an array of transfer functions.

`sys = getIOTransfer(sllin,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(sllin,ios)` returns the transfer function for the inputs and outputs specified by `ios` for the model associated with the `slLinearizer` interface, `sllin`. 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 `sllin`.

`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.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

in - Input analysis point signal name

string | cell array of strings

Input analysis point signal name, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 `in` as the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

  You can specify `in` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `in` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

  For example, `in = 'y1m'`.

- Cell array of strings — Specifies multiple analysis point names. For example, `in = {'y1m','y2m'}`.

out - Output analysis point signal name

string | cell array of strings

Output analysis point signal name, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 `out` as the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

  You can specify `out` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `out` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

  For example, `out = 'y1m'`.

- Cell array of strings — Specifies multiple analysis point names. For example, `out = {'y1m','y2m'}`.

temp_opening - Temporary opening signal name

string | cell array of strings

Temporary opening signal name, specified as:

- String — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `sllin`. To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 `temp_opening` as the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

You can specify `temp_opening` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

For example, `temp_opening = 'y1m'`.

- Cell array of strings — 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 `sllin`. To view the list of analysis points, type `sllin`. To use a point that is not in the list of analysis points for `sllin`, 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 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 = sllin.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 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 = sllin.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

`ss` model

Transfer function for specified I/Os, returned as described below:

- If you did not configure `sllin.Parameters` and `sllin.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 an `ss` model
- If you configured `sllin.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as an `ss` model array of the same size as `sllin.OperatingPoints`.
- If you configured `sllin.Parameters` and specified `sllin.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 an `ss` model array of the same size as the parameter grid.

- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `sllin.OperatingPoints` is the same size as the parameter grid specified by `sllin.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as an `ss` model array of the same size as the parameter grid
- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as an `ss` model array of size `N`-by-`p`.

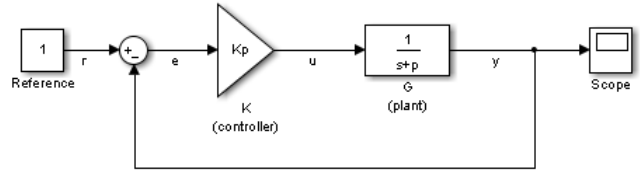
Examples

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`.

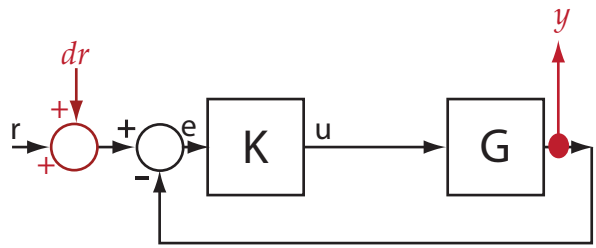
```
addPoint(sllin,{'r','y'});
```

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`.



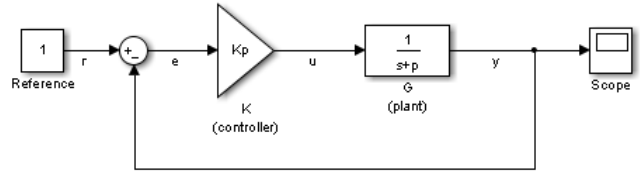
`sys` is the transfer function from `dr` to `y`, which is equal to $(I+GK)^{-1}GK$.

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`.

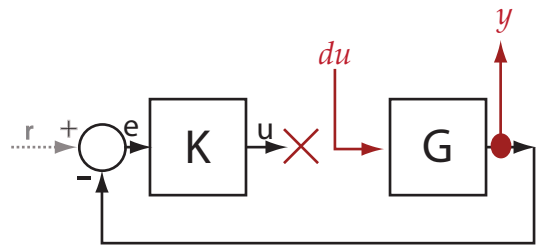
```
addPoint(sllin,{'u','y'});
```

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.



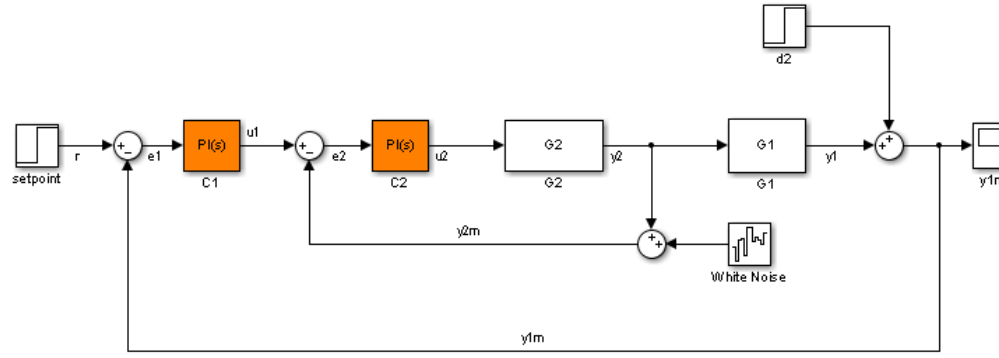
`sys` is the transfer function from `du` to `y`, which is equal to G.

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_ra
```

Obtain the open-loop transfer function from `e2` to `y2`.

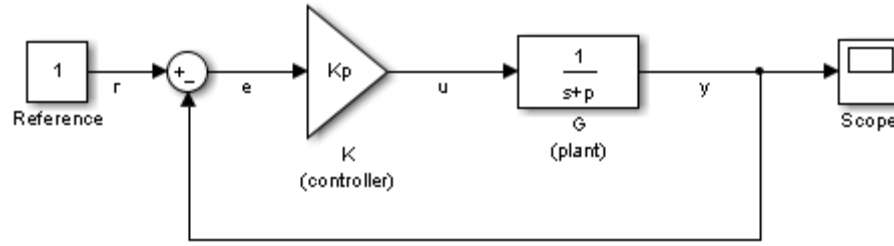
```
sys = getIOTransfer(sllin,'e2','y2','e2',mdl_index);
```

Definitions

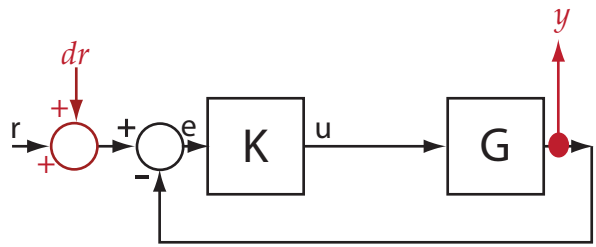
Transfer Functions

A *transfer function* is an LTI system's response at a linearization output point to a linearization input. You perform linear analysis on transfer functions to understand the stability, time-domain 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:



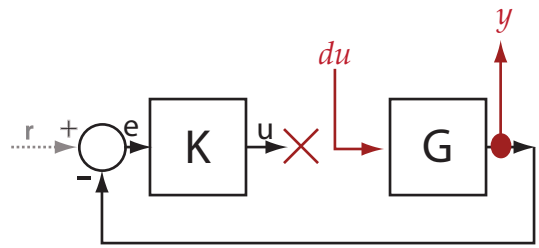
You can calculate the transfer function from the reference input signal to the plant output signal. The *reference input* (also referred to as *setpoint*) , `r`, originates at the Reference block, and the *plant output*, `y`, originates at the G block. This transfer function is also called the *overall 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.

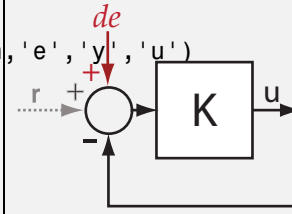
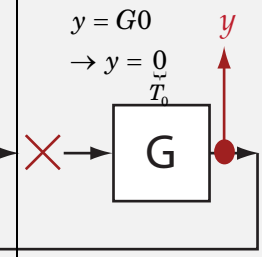
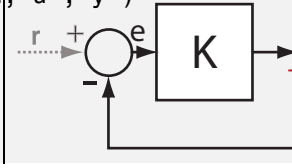
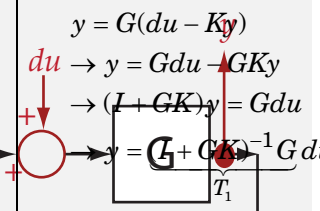
Similarly, to obtain the *controller transfer function*, calculate the transfer function from the controller input, `e`, to the controller output, `u`. Break the feedback loop at `y`, `e`, or `u`.

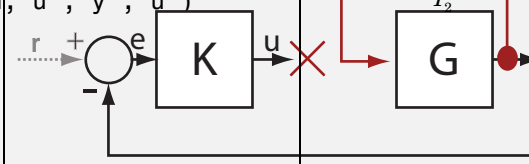
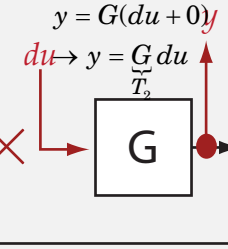
You can use `getIOTransfer` to obtain a variety of open-loop and closed-loop transfer functions. To configure the transfer function, specify analysis points as inputs, outputs, and openings (temporary or permanent), in any combination. The software treats each combination uniquely. Consider the following code that shows some different ways that you can use the analysis point, `u`, to obtain a transfer function:

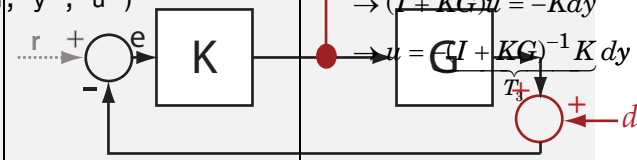
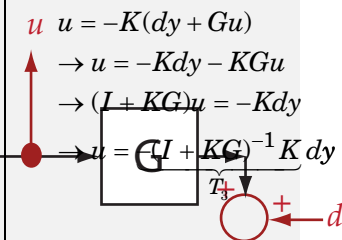
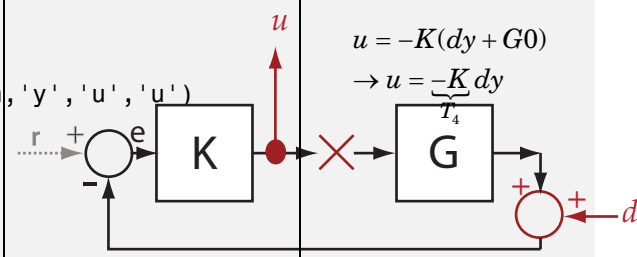
```
sllin = slLinearizer('ex_scd_simple_fdbk')

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');
```

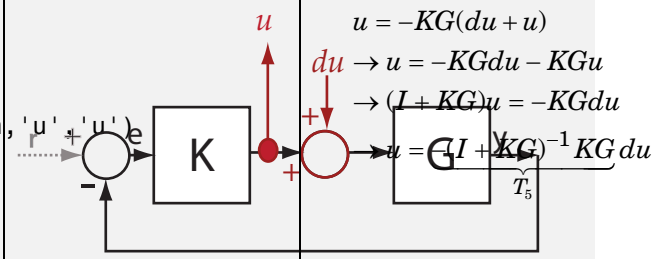
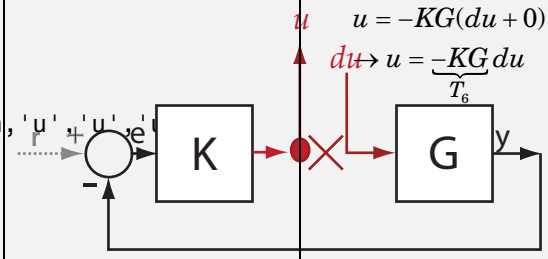
In `T0`, `u` specifies a loop break. In `T1`, `u` specifies only an input, whereas in `T2`, `u` specifies an input and an opening, also referred to as an *open-loop input*. In `T3`, `u` specifies only an output, whereas in `T4`, `u`

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 the 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`.	
Input Example code: `T1 = getIOTransfer(sllin,'u','y')`		

u Specifies...	How getIOTransfer Treats the Analysis Points	Transfer Function
	The software adds a linearization input, `du`, at `u`, and a linearization output at `y`.	
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`.	

u Specifies...	How getIOTransfer Treats the Analysis Points	Transfer Function
Output Example code: `T3 = getIOTransfer(sllin,'y','u')`	 The software adds a linearization input, `dy`, at `y` and a linearization output at `u`.	
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 Specifies...	How getIOTransfer Treats the Analysis Points	Transfer Function
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`.	$u = -KG(du + u)$ $du \rightarrow u = -KGdu - KGu$ $\rightarrow (I + KG)u = -KGdu$ $u = -(I + KG)^{-1} KG\, du$ T_5
Loop transfer function point Example code: `T6 = getIOTransfer(sllin,'u','u')` **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`.	$u = -KG(du + 0)$ $du \rightarrow u = -KG\, du$ T_6

The software does not modify the Simulink model when it computes the transfer function.

Analysis Points

Analysis points, used by the `slLinearizer` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addPoint | addOpening | getLoopTransfer | getSensitivity | getCompSensitivity`

Related Examples

- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "How the Software Treats Loop Openings" on page 2-158

Purpose

Open-loop transfer function at specified point using `slLinearizer` interface

Syntax

```
sys = getLoopTransfer(sllin,pt)
sys = getLoopTransfer(sllin,pt,sign)

sys = getLoopTransfer(sllin,pt,temp_opening)
sys = getLoopTransfer(sllin,pt,temp_opening,sign)

sys = getLoopTransfer( ___ ,mdl_index)
```

Description

`sys = getLoopTransfer(sllin,pt)` returns the point-to-point open-loop transfer function at the specified analysis point for the model associated with the `slLinearizer` interface, `sllin`.

The software enforces all the permanent loop openings specified for `sllin` when it calculates `sys`. If you configured either `sllin.Parameters`, or `sllin.OperatingPoints`, or both, `getLoopTransfer` performs multiple linearizations and returns an array of loop transfer functions.

`sys = getLoopTransfer(sllin,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(sllin,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(sllin,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.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Analysis point signal name

string | cell array of strings

Analysis point signal name, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

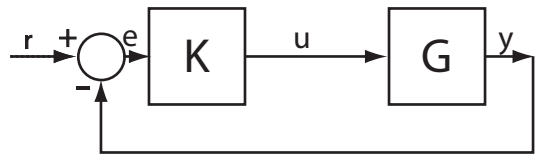
  You can specify `pt` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as

`'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

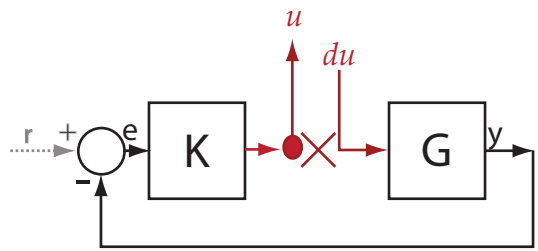
For example, `pt = 'y1m'`.

- Cell array of strings — 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:

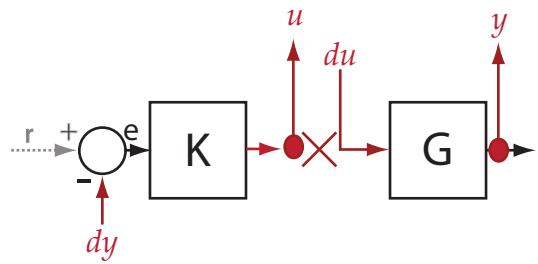


Specify `pt` as `'u'`.



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

string | cell array of strings

Temporary opening signal name, specified as:

- String — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `sllin`. To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 `temp_opening` as

the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

You can specify `temp_opening` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

For example, `temp_opening = 'y1m'`.

- Cell array of strings — 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 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 = sllin.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 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 = sllin.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

`ss` object

Point-to-point open-loop transfer function, returned as described below:

- If you did not configure `sllin.Parameters` and `sllin.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 an `ss` model
- If you configured `sllin.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as an `ss` model array of the same size as `sllin.OperatingPoints`.
- If you configured `sllin.Parameters` and specified `sllin.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 an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `sllin.OperatingPoints` is the same size as the parameter grid specified by `sllin.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as an `ss` model array of the same size as the parameter grid

- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as an `ss` model array of size `N`-by-`p`.

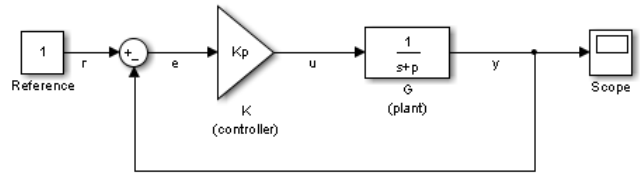
Examples

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.

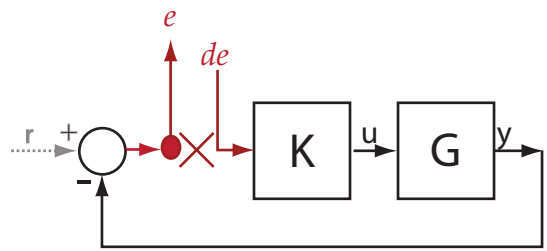
```
addPoint(sllin,'e');
```

Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e');
tf(sys)
```

```
  From input "e" to output:
   -3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



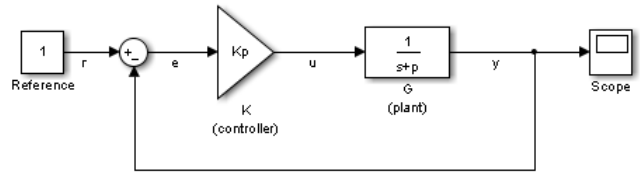
`sys` is the transfer function from `de` to `e`. Because the software assumes positive-feedback, it returns `sys` as $-GK$.

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.

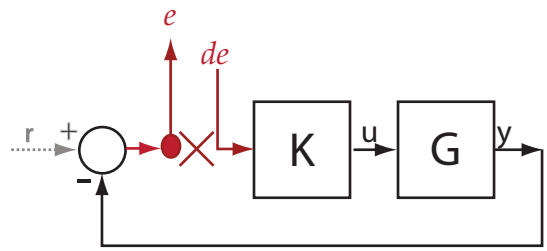
```
addPoint(sllin,'e');
```

Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e',-1);
tf(sys)
```

```
ans =

  From input "e" to output:
    3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



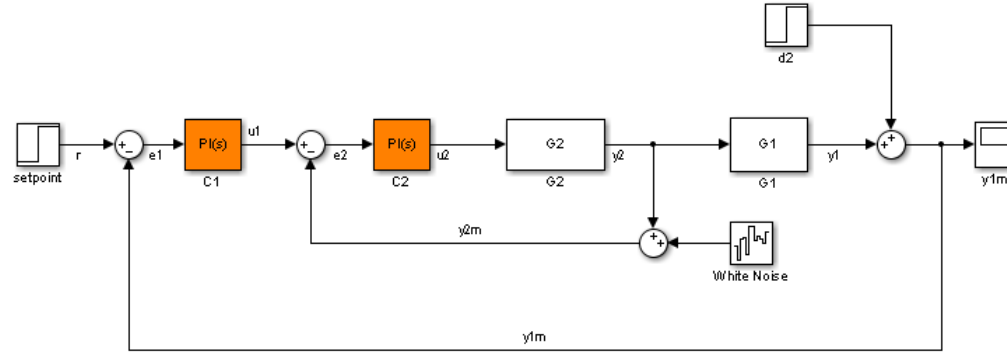
`sys` is the transfer function from `de` to `e`. Because the third input argument indicates negative-feedback, the software returns `sys` as *GK*.

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');
```

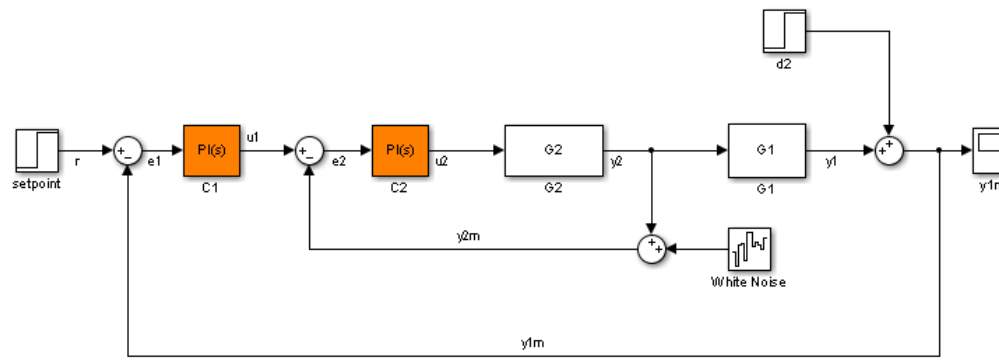
Here, `'y1m'`, the third input argument, specifies a temporary loop opening. The software assumes positive-feedback when it calculates `sys`.

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_ra
```

Obtain the inner-loop loop transfer function at `e2`, with the outer loop open.

```
sys = getLoopTransfer(sllin,'e2','y1m',-1,mdl_index);
```

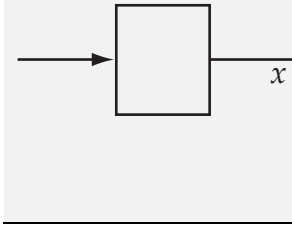
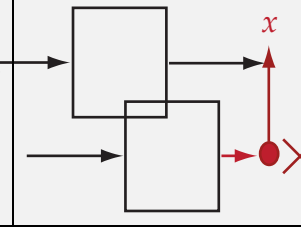
The fourth input argument specifies negative-feedback for the loop transfer calculation.

Definitions

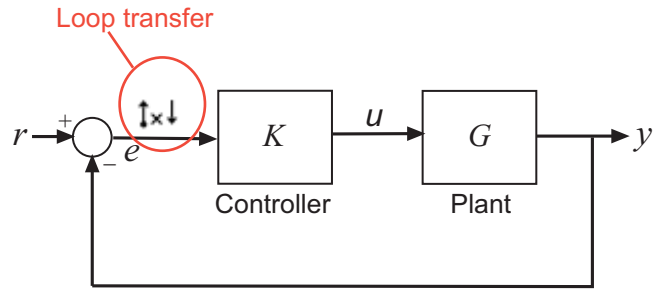
Loop Transfer Function

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 the 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` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow

for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer` | `addPoint` | `addOpening` | `getIOTransfer` | `getSensitivity` | `getCompSensitivity`

Related Examples

- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "How the Software Treats Loop Openings" on page 2-158

getSensitivity

Purpose

Sensitivity function at specified point using `slLinearizer` interface

Syntax

```
sys = getSensitivity(sllin,pt)
sys = getSensitivity(sllin,pt,temp_opening)
sys = getSensitivity( ___ ,mdl_index)
```

Description

`sys = getSensitivity(sllin,pt)` returns the sensitivity function at the specified analysis point for the model associated with the `slLinearizer` interface, `sllin`.

The software enforces all the permanent openings specified for `sllin` when it calculates `sys`. If you configured either `sllin.Parameters`, or `sllin.OperatingPoints`, or both, `getSensitivity` performs multiple linearizations and returns an array of sensitivity functions.

`sys = getSensitivity(sllin,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.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Analysis point signal name

string | cell array of strings

Analysis point signal name, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

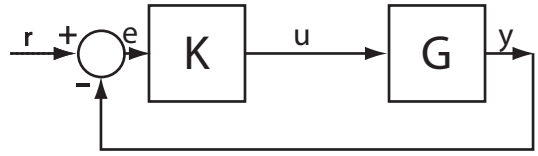
  You can specify `pt` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

  For example, `pt = 'y1m'`.

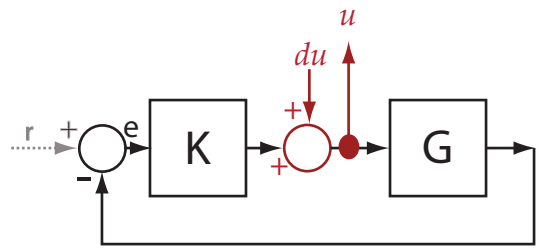
- Cell array of strings — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `sys`, the software adds a linearization input, followed by a linearization output at `pt`.

Consider the following model:

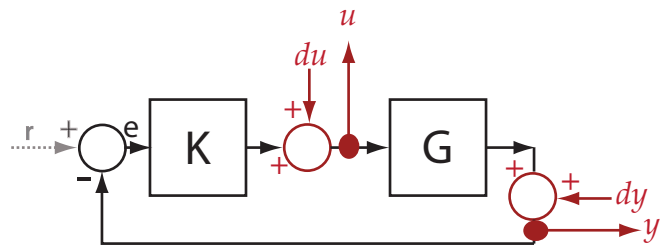


Specify `pt` as `'u'`:



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

string | cell array of strings

Temporary opening signal name, specified as:

- String — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `sllin`. To determine the signal name associated with an analysis point, type `sllin`. The software displays the

contents of `sllin` 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 `temp_opening` as the block name. To use a point not in the list of analysis points for `sllin`, first add the point using `addPoint`.

You can specify `temp_opening` as a uniquely matching substring of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a substring of the signal name for any other analysis point of `sllin`.

For example, `temp_opening = 'y1m'`.

- Cell array of strings — 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 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 = sllin.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 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 = sllin.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

`ss` model

Sensitivity function, returned as described below:

- If you did not configure `sllin.Parameters` and `sllin.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 an `ss` model
- If you configured `sllin.Parameters` only, the software computes a linearization for each parameter grid point. `sys` is returned as an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.OperatingPoints` only, the software computes a linearization for each specified operating point. `sys` is returned as an `ss` model array of the same size as `sllin.OperatingPoints`.
- If you configured `sllin.Parameters` and specified `sllin.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 an `ss` model array of the same size as the parameter grid.
- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter

grid point. The software requires that `sllin.OperatingPoints` is the same size as the parameter grid specified by `sllin.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `sys` is returned as an `ss` model array of the same size as the parameter grid

- If you configured `sllin.Parameters` and specified `sllin.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose you specify a parameter grid of size `p` and `N` snapshot times. `sys` is returned as an `ss` model array of size `N`-by-`p`.

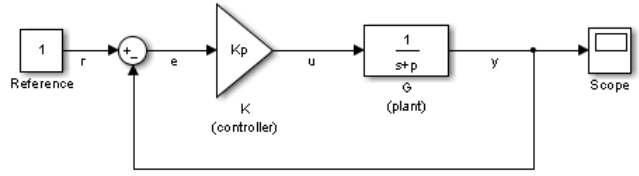
Examples

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`.

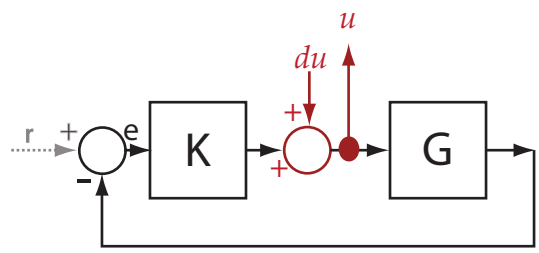
```
addPoint(sllin,'u');
```

Obtain the sensitivity at the plant input, `u`.

```
sys = getSensitivity(sllin,'u');
tf(sys)
```

```
  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`.



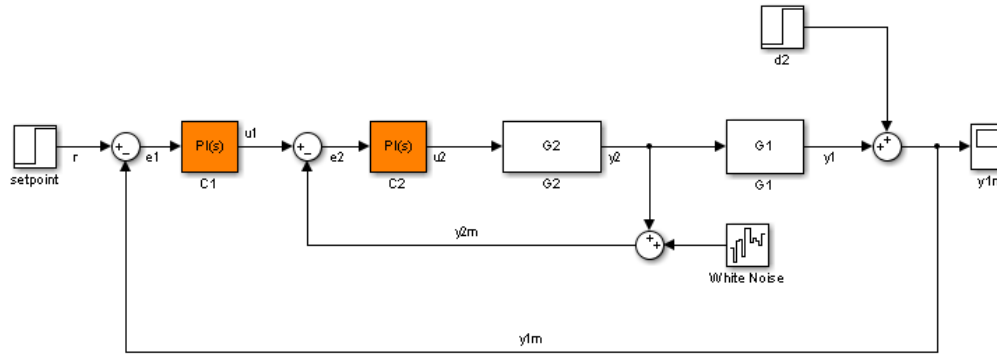
`sys` is the transfer function from `du` to `u`, which is equal to $(I+KG)^{-1}$.

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');
```

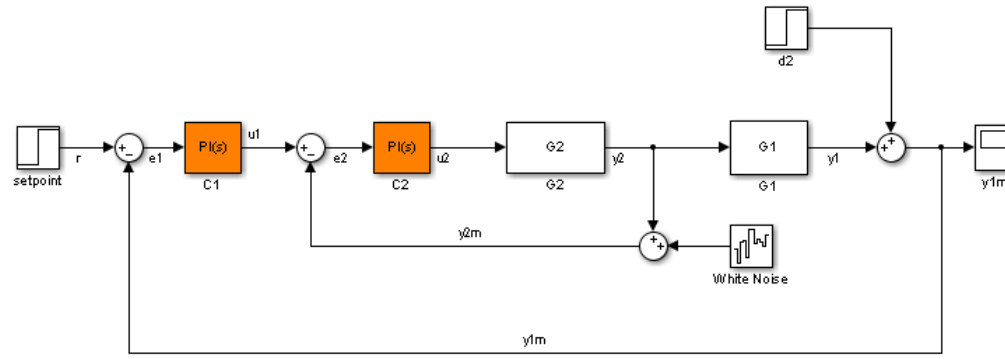
Here, `'y1m'`, the third input argument, specifies a temporary opening of the outer loop.

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
```

Obtain the sensitivity at the output of `G2` for the specified parameter combination.

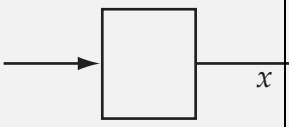
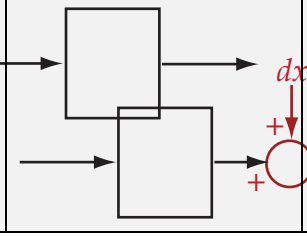
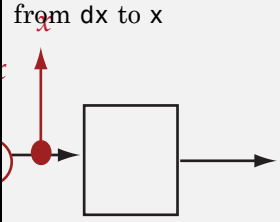
```
sys = getSensitivity(sllin,'y2','y1m',mdl_index);
```

Definitions

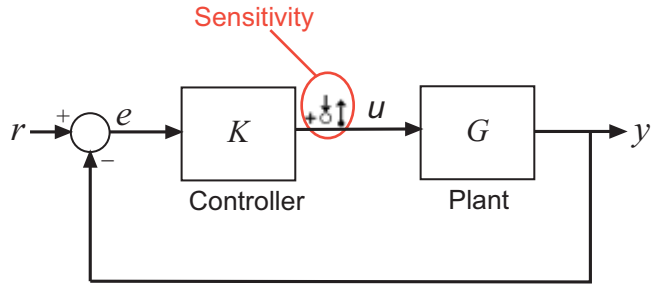
Sensitivity Function

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 the 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` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addPoint | addOpening | getIOTransfer | getLoopTransfer | getCompSensitivity`

Related Examples

- "Vary Parameter Values and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-169
- "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer" on page 2-177
- "Use Response Plots to Analyze Batch Linearization Results" on page 2-183

Concepts

- "How the Software Treats Loop Openings" on page 2-158

Purpose

Resynchronize `slLinearizer` interface with current model state

Syntax

```
refresh(sllin)
```

Description

`refresh(sllin)` resynchronizes the `slLinearizer` interface with the current state of the model. The interface recompiles the model for the next call to `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, or `getCompSensitivity` . This model recompilation ensures that the interface uses the current model state when computing linearizations.

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.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

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);
```

See Also

`slLinearizer` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity`

Purpose

Remove all openings from list of permanent openings in `slLinearizer` interface

Syntax

```
removeAllOpenings(sllin)
```

Description

`removeAllOpenings(sllin)` removes all openings from the list of permanent openings in the `slLinearizer` interface, `sllin`. This function does not modify the model associated with `sllin`.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

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 addPoint method to add new points.
No permanent openings. Use addOpening method to add new permanent opening
Other properties (with dot notation get/set access):
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`slLinearizer | addOpening | removeOpening`

Purpose

Remove all points from list of analysis points in `slLinearizer` interface

Syntax

```
removeAllPoints(sllin)
```

Description

`removeAllPoints(sllin)` removes all points from the list of analysis points for the `slLinearizer` interface, `sllin`. This function does not modify the model associated with `sllin`.

Input Arguments

sllin - Interface to linearize a Simulink model

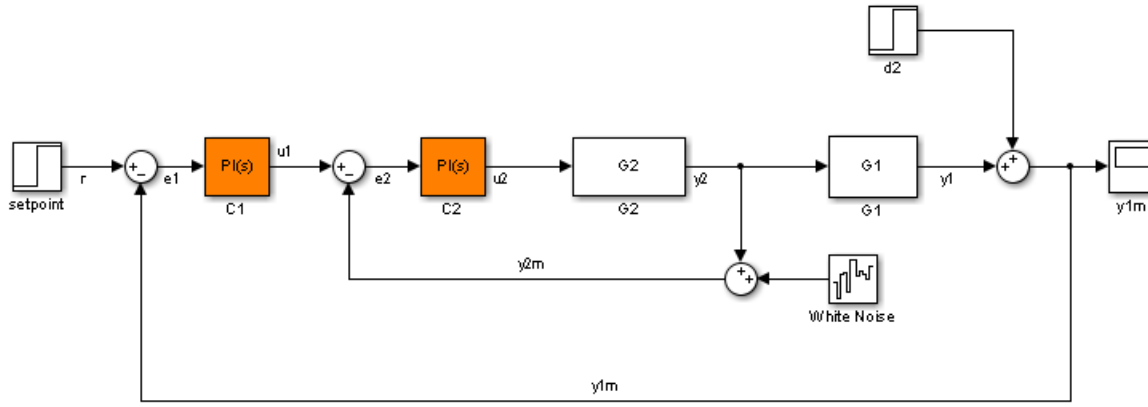
`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

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'})
```

```
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 addOpening method to add new permanent opening
Other properties (with dot notation get/set access):
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

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 addPoint method to add new points.
No permanent openings. Use addOpening method to add new permanent opening
```

```
Other properties (with dot notation get/set access):
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

See Also

`removePoint` | `slLinearizer` | `addPoint`

removeOpening

Purpose

Remove opening from list of permanent loop openings in `slLinearizer` interface

Syntax

```
removeOpening(sllin,op)
```

Description

`removeOpening(sllin,op)` removes the specified opening, `op`, from the list of permanent openings for the `slLinearizer` interface, `sllin`. You can specify `op` to remove either a single or multiple openings.

`removeOpening` does not modify the model associated with `sllin`.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

op - Opening

string | cell array of strings | positive integer | vector of positive integers

Opening to remove from the list of permanent openings for `sllin`, specified as:

- String — Opening signal name.

  To determine the signal name associated with an opening, type `sllin`. The software displays the contents of `sllin` 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 substring 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 substring of the signal name for any other opening of `sllin`.

  For example, `op = 'y1m'`.

- Cell array of strings — Specifies multiple opening names. For example, `op = {'y1m','y2m'}`.
- Positive integer — Opening index.

  To determine the index of an opening, type `sllin`. The software displays the contents of `sllin` 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]`.

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 addPoint method 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

Other 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 addPoint method 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

Other 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]);
```

Definitions

Permanent Openings

Permanent openings, used by the `slLinearizer` interface, identify locations within a model where the software breaks the signal flow for linearization. The software enforces these openings for every linearization. Use permanent openings to isolate a specific model component for the purposes of linearization. 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 from your linearizations. 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` interface when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removePoint` command.

To view all the openings of `sllin`, type `sllin` at the command prompt to display the interface contents. For each permanent opening of `sllin`, the display includes the block name and port number and the name of the signal that originates at this location.

See Also

`removeAllOpenings` | `addOpening` | `slLinearizer` | `removePoint`

removePoint

Purpose

Remove point from list of analysis points in `slLinearizer` interface

Syntax

```
removePoint(sllin,pt)
```

Description

`removePoint(sllin,pt)` removes the specified point, `pt`, from the list of analysis points for the `slLinearizer` interface, `sllin`. You can specify `pt` to remove either a single or multiple points.

`removePoint` does not modify the model associated with `sllin`.

Input Arguments

sllin - Interface to linearize a Simulink model

`slLinearizer` interface

Interface to linearize a Simulink model, specified as an `slLinearizer` interface.

pt - Analysis point

string | cell array of strings | positive integer | vector of positive integers

Analysis point to remove from the list of analysis points for `sllin`, specified as:

- String — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `sllin`. The software displays the contents of `sllin` 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 substring 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 substring of the signal name for any other analysis point of `sllin`.

  For example, `pt = 'y1m'`.

- Cell array of strings — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.
- Positive integer or — Analysis point index.

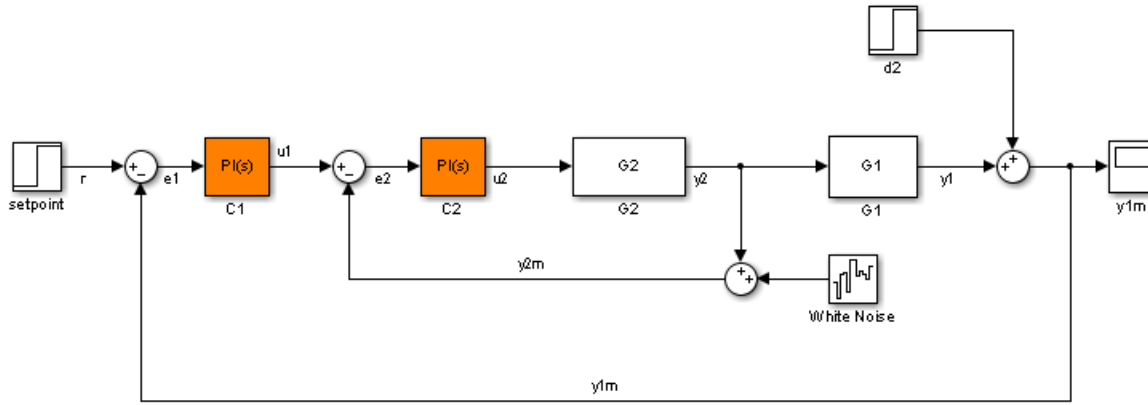
  To determine the index of an analysis point, type `sllin`. The software displays the contents of `sllin` in the MATLAB command window, including the analysis points indices.

  For example, `pt = 1`.
- Vector of positive integers — Specifies multiple analysis point indices. For example, `pt = [1 2]`.

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.



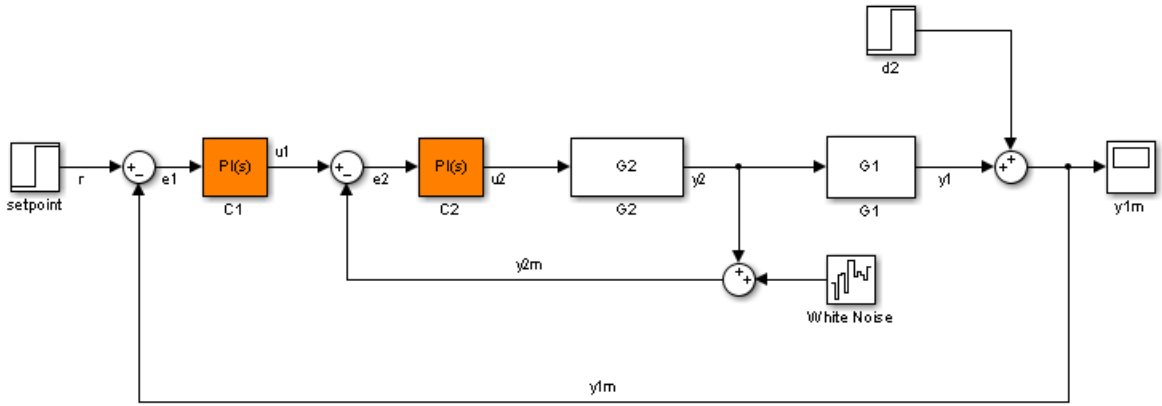

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

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.



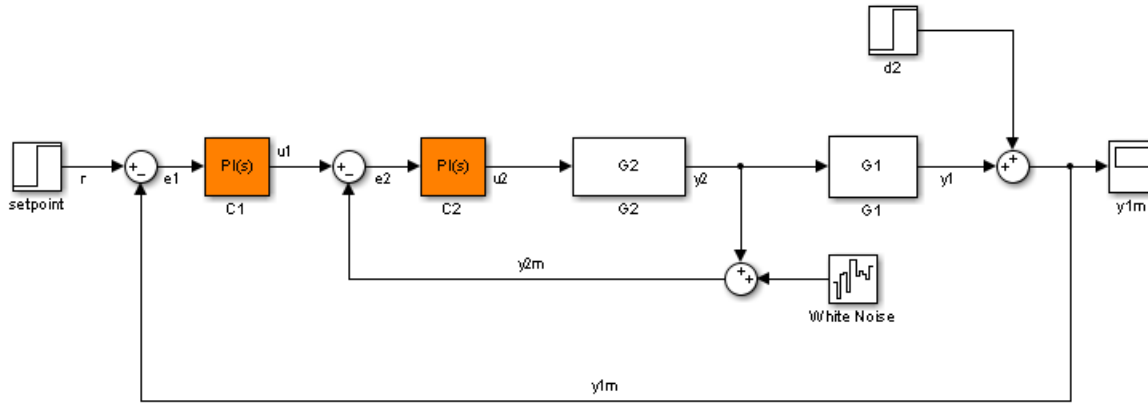

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

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 addOpening method to add new permanent opening
Other properties (with dot notation get/set access):
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

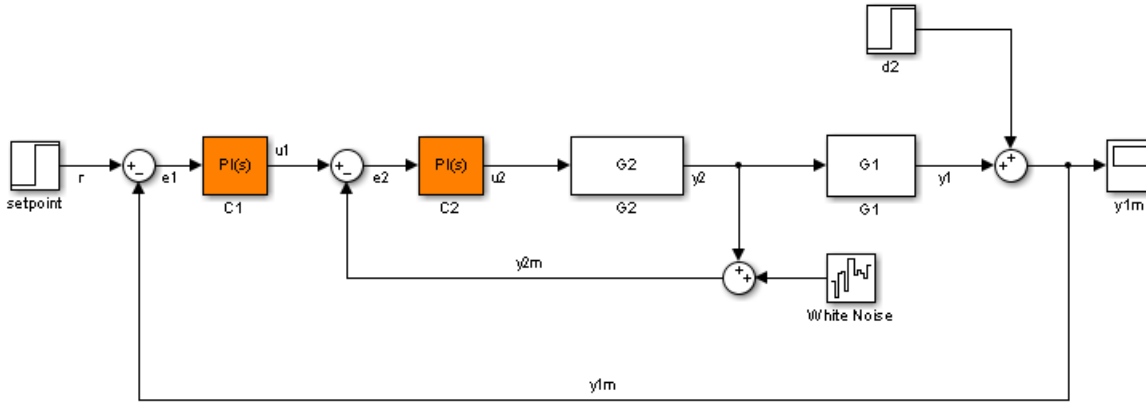
The displays shows that `y1m` is the third analysis point of `sllin` .

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 addOpening method to add new permanent oper
Other 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]);
```

Definitions

Analysis Points

Analysis points, used by the `slLinearizer` interface, identify locations within a model that are relevant for linear analysis. You use analysis points as inputs to the linearization commands of the `slLinearizer` interface: `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.

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 analysis point.

You can add analysis points to an `slLinearizer` interface, `sllin`, when you create the interface:

```
sllin = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `sllin`, type `sllin` at the command prompt to display the interface contents. For each analysis point of `sllin`, the display includes the block name and port number and the name of the signal that originates at this point.

See Also

`removeAllPoints` | `addPoint` | `slLinearizer` | `removeOpening`

Purpose

Add block to list of tuned blocks

Syntax

```
addBlock(ST,blockID)
```

Description

`addBlock(ST,blockID)` adds the block referenced by `blockID` to the `TunedBlocks` property of the `slTunable` interface `ST`. The `TunedBlocks` property lists the tunable blocks in the control system described by `ST`.

Input Arguments

ST

`slTunable` interface describing the control system to be tuned.

blockID

String or cell array of strings identifying one or more blocks to add to the `TunedBlocks` property of `ST`. You can specify blocks with an abbreviated block name provided that the string matches the end of the full block path and unambiguously identifies the block to add.

Examples

Create an `slTunable` interface for the Simulink model `rct_cascade` and add a block to the list of tuned blocks in the interface.

```
open_system('rct_cascade')
ST0 = slTunable('rct_cascade','C1');
addBlock(ST0,'C2');
```

See Also

`slTunable` | `slTunable.addMeasurement` | `slTunable.addIO` | `slTunable.addControl`

slTunable.addControl

Purpose

Add control signal to list of control signals in `slTunable` interface

Syntax

```
addControl(ST,signalID)
```

Description

`addControl(ST,signalID)` adds the signals referenced by `signalID` to the `Controls` property of the `slTunable` interface `ST`. The `Controls` signals are generated by the controller and drive the actuators in the control system described by `ST`. The `Controls` and `Measurements` signals together define the boundary between the plant and controller subsystems for tuning with `slTunable.looptune`. You can also use `Controls` signals as inputs, outputs, or loop-opening locations when specifying `TuningGoal` requirements for control system tuning.

Input Arguments

ST

`slTunable` interface describing the control system to be tuned.

signalID

String or cell array of strings identifying one or more signals to add to the `Controls` property of `ST`. Each string can be one of the following:

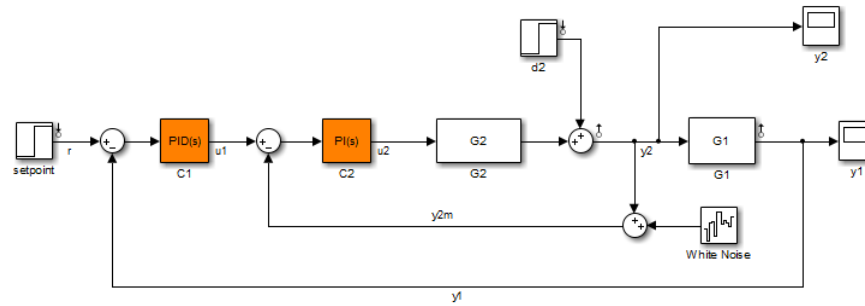
- Signal name, for example `'torque'`
- Block path for a block with a single output port, for example `'Motor/PID'`
- Path to block and port originating the signal, for example `'Engine Model/1'` or `'Engine Model/torque'`

Alternatively, you can specify `signalID` as a linearization I/O object or vector of linearization I/O objects (see `linio`).

Examples

Specify Control Signal in Control System to Be Tuned

This example shows how to specify control signals in an `slTunable` interface for tuning the Simulink model `rct_cascade`:



Create an `slTunable` interface for the model.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);
```

Specify the control signal for the control system.

```
addControl(ST0,'u2');
```

In `rct_cascade`, the control signal (the actuator signal) is `u2`, the output of the inner-loop controller.

See Also

`slTunable` | `slTunable.addMeasurement` | `slTunable.addIO` | `slTunable.addOpening` | `slTunable.addSwitch`

slTunable.addIO

Purpose

Add I/O points list of linearization I/Os in `slTunable` interface

Syntax

```
addIO(ST,signalID,type)
addIO(ST,iopoint)
```

Description

`addIO(ST,signalID,type)` adds the signal locations referenced by `signalID` to the `IOs` property of the `slTunable` interface `ST`. The `type` argument specifies whether each I/O type is input, output, or both.

`addIO(ST,iopoint)` reads the signal path and I/O type from the linearization I/O point object `iopoint`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

signalID

String or cell array of strings identifying one or more signals to add to the `IOs` property of `ST`. Each string can be one 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'`

type

String specifying the I/O type of `signalID`. The type can be one of the following:

- `'input'` — Input perturbation. Specifies an additive input to the signal.
- `'output'` — Output measurement. Takes measurement at a signal.

- `'sensitivity'` — Sensitivity. Specifies an additive input followed by an output measurement.
- `'compsensitivity'` — Complementary sensitivity. Specifies an output followed by an additive input.

If `signalID` is an array of signal names, `type` is an array of the same length. The *k*th entry in `type` specifies the type of the *k*th entry in `signalID`.

For more information about linearization I/O point types, see `linio`.

iopoint

Linearization I/O object or vector of linearization I/O objects (see `linio`).

Examples

Create a `slTunable` interface for tuning the Simulink model `rct_cascade`. Add an input perturbation at the signal `u1`, and a sensitivity point at the signal `y2m`.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);

addIO(ST0,'u1','input');
addIO(ST0,'y2m','sensitivity');
```

See Also

`slTunable` | `slTunable.addMeasurement` | `slTunable.addControl` | `slTunable.addOpening` | `slTunable.addSwitch` | `linio`

How To

- “Specifying Subsystem, Loop, or Block to Linearize” on page 2-13

slTunable.addMeasurement

Purpose

Add measurement signal to list of measurement signals in `slTunable` interface

Syntax

```
addMeasurement(ST,signalID)
```

Description

`addMeasurement(ST,signalID)` adds the signals referenced by `signalID` to the `Measurements` property of the `slTunable` interface `ST`. The `Measurements` signals are generated by the plant and used by the controller in the control system described by `ST`. The `Controls` and `Measurements` signals together define the boundary between the plant and controller subsystems for tuning with `slTunable.looptune`. You can also use `Measurements` signals as inputs, outputs, or loop-opening locations when specifying `TuningGoal` requirements for control system tuning.

Input Arguments

ST

`slTunable` interface describing the control system to be tuned.

signalID

String or cell array of strings identifying one or more signals to add to the `Measurements` property of `ST`. Each string can be one of the following:

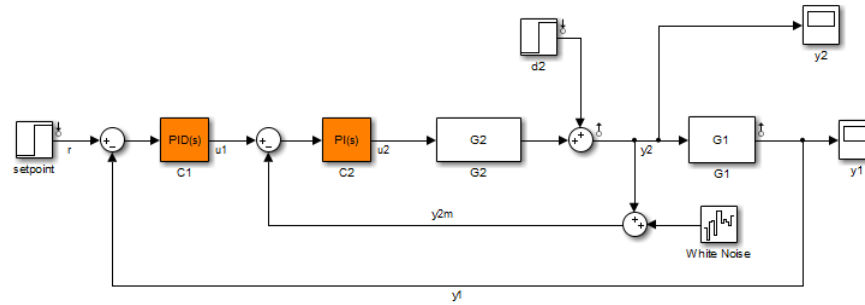
- Signal name, for example `'speed'`
- Block path for a block with a single output port, for example `'"Engine/SpeedSensor'`
- Path to block and port originating the signal, for example `'Engine Model/1'` or `'Engine Model/speed'`

Alternatively, you can specify `signalID` as a linearization I/O object or vector of linearization I/O objects (see `linio`).

Examples

Specify Measurement Signals in Control System to Be Tuned

This example shows how to specify measurement signals in an `slTunable` interface for tuning the Simulink model `rct_cascade`:



Create an `slTunable` interface for the model.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);
```

Specify the measurement signal for the control system.

```
addMeasurement(ST0,{'y1','y2m'});
```

In `rct_cascade`, the measurement signals are `y1` and `y2m`, each of which feeds into a portion of the cascaded controller.

See Also

`slTunable` | `slTunable.addControl` | `slTunable.addIO` | `slTunable.addOpening` | `slTunable.addSwitch`

slTunable.addOpening

Purpose

Add new opening location to list of loop openings in `slTunable` interface

Syntax

```
addOpening(ST,signalID)
```

Description

`addOpening(ST,signalID)` adds the signal locations referenced by `signalID` to the `Openings` property of the `slTunable` interface `ST`. Loop openings are enforced at `Openings` locations for all tuning and analysis commands involving `ST`.

Tips

- To add a location where a loop opening is optionally enforced for a particular analysis or tuning requirement, use `slTunable.addSwitch`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

signalID

String or cell array of strings identifying one or more signals to add to the `Openings` property of `ST`. Each string can be one of the following:

- Signal name, for example `'u'`
- Block path for a block with a single output port, for example `'Motor/PID'`
- Path to block and port originating the signal, for example `'Controller/1'` or `'Controller/u'`

Examples

Create a `slTunable` interface for tuning the Simulink model `rct_cascade`. Add a loop opening location at the signal `y1`.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);

addOpening(ST0,'y1');
```

The loop opening at `y1` is enforced for all tuning and analysis commands involving `ST0`.

See Also

`slTunable` | `slTunable.addMeasurement` | `slTunable.addIO` | `slTunable.addControl` | `slTunable.addSwitch`

slTunable.addSwitch

Purpose

Add switch location to list of loop opening switches in `slTunable` interface

Syntax

```
addSwitch(ST,signalID)
```

Description

`addSwitch(ST,signalID)` adds the locations of the signals referenced by `signalID` to the `Switches` property of the `slTunable` interface `ST`. Switches are locations where you can optionally open a loop for tuning and analysis commands involving `ST`. You can also use `Switches` signals as inputs, outputs, or loop-opening locations when specifying `TuningGoal` requirements for control system tuning.

Tips

- To add a location where a loop opening is always enforced for all analysis or tuning requirement, use `slTunable.addOpening`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

signalID

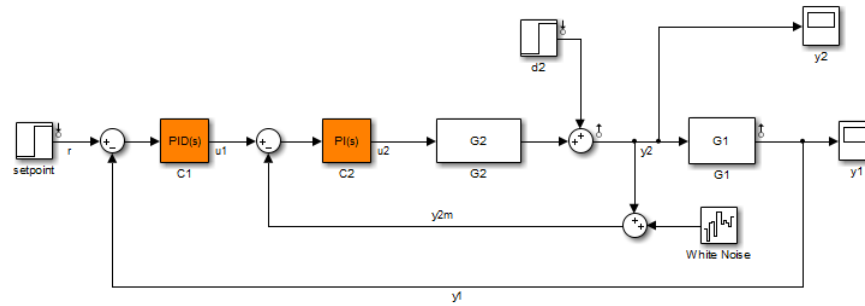
String or cell array of strings identifying one or more signals to add to the `Switches` property of `ST`. Each string can be one of the following:

- Signal name, for example `'u'`
- Block path for a block with a single output port, for example `'Motor/PID'`
- Path to block and port originating the signal, for example `'Controller/1'` or `'Controller/u'`

Examples

Specify Switch Location in Control System to Be Tuned

This example shows how to specify a location for an optional loop opening (a switch) in an `slTunable` interface for tuning the Simulink model `rct_cascade`:



Create an `slTunable` interface for the model.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);
```

Specify the switch location.

```
addSwitch(ST0,'y1');
```

When tuning a cascade control system, it can be useful to impose certain requirements on the inner loop that are enforced with the outer loop open. However, such a control system also has overall requirements are enforced with all loops closed. Adding a switch instead of a loop opening at `y1` allows you to compute some responses with the outer loop open, and other responses with the outer loop closed.

See Also

`slTunable` | `slTunable.addMeasurement` | `slTunable.addIO` | `slTunable.addControl` | `slTunable.addOpening`

slTunable.getBlockParam

Purpose

Get parametrization of a particular tuned block in `slTunable` interface

Syntax

```
blockparam = getBlockParam(ST,block)
```

Description

`blockparam = getBlockParam(ST,block)` returns the parametrization used to tune the Simulink block, `block`.

Tips

The `slTunable` interface automatically associates a parametric model with each Simulink block listed in the `TunedBlocks` property of `ST`. This parametrization expresses each tunable block as a Control Design Block or a tunable `genss` model. The parametrization specifies which parameters are tuned by commands such as `slTunable.systune` or `slTunable.looptune`. Use `getBlockParam` to access this parametrization. The tunable parameters of `ST` are the union of the parametrizations of all tunable blocks in `ST`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

Output Arguments

blockparam

Parametrization of `block`. The parametrization is either:

- A tunable Control Design Block.
- A tunable `genss` model, if you specified a parametrization for `block` using `slTunable.setBlockParam`.
- An empty array (`[]`), if `slTunable` cannot parametrize `block`. You can use `slTunable.setBlockParam` to specify a parametrization for such blocks.

Examples

Create an `slTunable` interface for the Simulink model `rct_cascade` and examine the block parametrization of one of the tunable blocks.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);

TC = getBlockParam(ST0,'C1')
```

These commands produce the result:

```
Parametric continuous-time PID controller "rct_cascade_C1" with formula:

           1            s
Kp + Ki * --- + Kd * --------
           s          Tf*s+1

and tunable parameters: Kp, Ki, Kd, Tf.
```

The block `C1` is a PID Controller block. Therefore, its parametrization in `ST0` is a `ltiblock.pid` (a Control Design Block).

See Also

`slTunable` | `slTunable.getBlockValue` | `slTunable.setBlockParam` | `genss` | `ltiblock.pid`

How To

- "Supported Blocks for Tuning in Simulink"

slTunable.getBlockRateConversion

Purpose

Query rate conversion method of tunable block

Syntax

```
method = getBlockRateConversion(ST,block)
[method,pwfreq] = getBlockRateConversion(ST,block)
```

Description

`method = getBlockRateConversion(ST,block)` returns the rate conversion method currently associated with a tunable block in an `slTunable` interface.

`[method,pwfreq] = getBlockRateConversion(ST,block)` also returns the prewarp frequency. When `method` is not `'tustin'`, the prewarp frequency is always zero.

Tips

- Use `getBlockRateConversion` to query the rate conversion method that the `slTunable` interface uses when converting the sampling time of the block parametrization.

  For example, you might use the `slTunable` interface to tune a mixed-rate Simulink model having a discrete-time controller and a continuous time plant. Tuning commands such as `looptune` or `systune` tune the model in continuous time. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, `writeBlockValue` automatically discretizes the tuned parametrization. It uses the block rate conversion method associated with the block for this discretization.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single tunable block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, if the string matches the end of the full block path and unambiguously identifies the block.

Output Arguments

method

Discretization method associated with the tunable block, returned as one of the following strings:

- `'zoh'` — Zero-order hold method
- `'tustin'` — Tustin method

pwfreq

Prewarp frequency for the Tustin method, returned as a positive scalar.

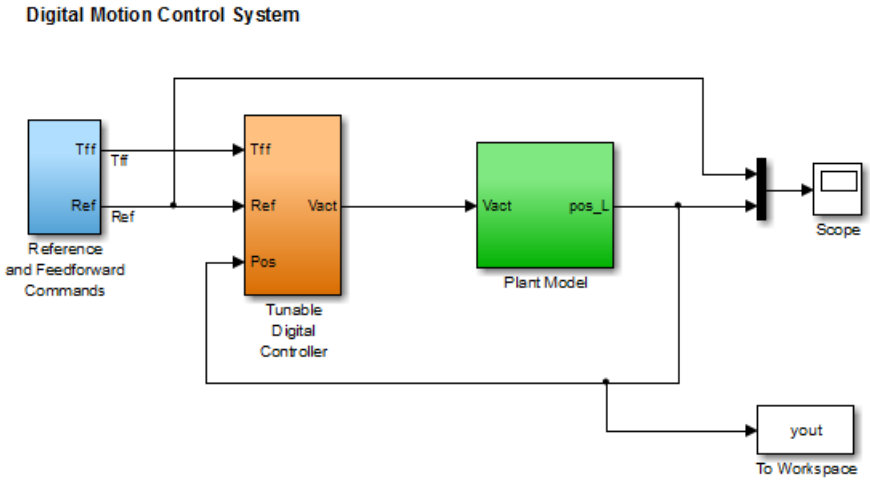
If the discretization method associated with the tunable block is zero-order hold or Tustin without prewarp, then `pwfreq` is 0.

Examples

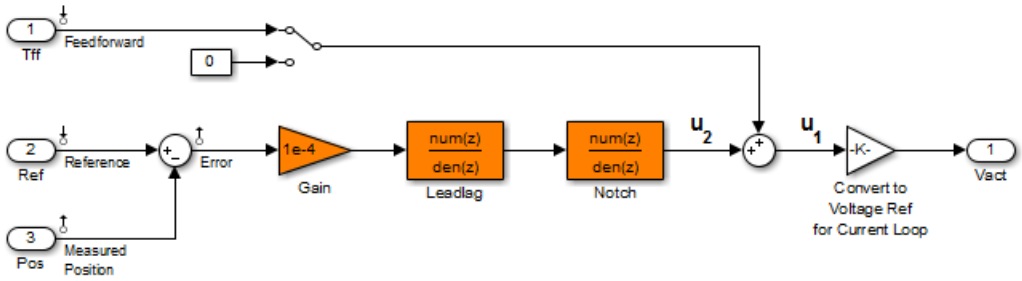
Create an `slTunable` interface describing the Simulink model `rct_dmcNotch`. Query and change the rate conversion method associated with a tunable block in the model.

Open the Simulink model.

```
open_system('rct_dmcNotch')
```



Double-click the `Tunable Digital Controller` subsystem to see the tunable blocks.



Create an `slTunable` interface for the model with the tunable blocks shown in orange.

```
ST = slTunable('rct_dmcNotch',{'Gain','Leadlag','Notch'});
```

The `LeadLag` and `Notch` blocks are discrete-time blocks. You might tune this control system in continuous time (for example, using `looptune` or

systune) and write the tuned parameters back to the Simulink model. If you do so, slTunable.writeBlockValue automatically discretizes the resulting tuned blocks using the discretization method associated with the Simulink block. Query the discretization method associated with the Notch block.

```
method = getBlockRateConversion(ST,'Notch')
```

```
method =

zoh
```

The result shows that the zero-order-hold (zoh) method is currently associated with the Notch block. This block represents a notch filter. Therefore, you may get better discretization results using the Tustin discretization method with a prewarp at the notch frequency. Assume that the tuned notch frequency is 340 rad/s.

```
wn = 340;
setBlockRateConversion(ST,'Notch','tustin',wn)
```

This command sets the rate conversion method of the Notch block to the Tustin method with a prewarp at the notch frequency. slTunable.writeBlockValue now discretizes the tuned block using this method.

For a more detailed treatment of this example, see "Tuning of a Digital Motion Control System".

See Also

slTunable | slTunable.writeBlockValue | slTunable.setBlockRateConversion

Tutorials

- "Tuning of a Digital Motion Control System"

How To

- "Continuous-Discrete Conversion Methods"

slTunable.getBlockValue

Purpose

Current value of a particular tuned block in `slTunable` interface

Syntax

```
val = getBlockValue(ST,block)
```

Description

`val = getBlockValue(ST,block)` returns the current value of the parametrization of the Simulink block `block`. The string `block` should designate one of the tuned blocks in the control system described by the `slTunable` interface `ST`.

Tips

- The `slTunable` interface automatically associates a parametric model with each Simulink block listed in the `TunedBlocks` property of `ST`. This parametrization expresses each tunable block as a Control Design Block or a tunable `genss` model. The parametrization specifies which parameters are tuned by commands such as `slTunable.looptune`. Use `getBlockValue` to access the value of this parametrization after tuning.
- The current value of the block parametrization in the `slTunable` interface `ST` can differ from the actual value of the corresponding block in the Simulink model. Use `slTunable.readBlockValue` or `slTunable.writeBlockValue` to align the value of the parametrization in `ST` with the actual block value in the model.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

Output Arguments

val

Current value of the parametrization of `block` in `ST`. The current value is a numeric LTI model such as `pid`, `ss`, or `tf`.

Examples

Create an `slTunable` interface for the Simulink model `rct_cascade` and examine the current value of the block parametrization of one of the tunable blocks.

```
open_system('rct_cascade')
tunedblocks = {'C1','C2'};
ST0 = slTunable('rct_cascade',tunedblocks);

val = getBlockValue(ST0,'C1')
```

These commands produce the result:

```
Continuous-time PI controller in parallel form:

           1
Kp + Ki * ---
           s

With Kp = 0.1, Ki = 0.1
```

The block `C1` is a PID Controller block. Therefore, the value of its parametrization in `ST0` is a `pid` controller model.

Use `getBlockValue` after tuning an `slTunable` interface with `slTunable.systune` or `slTunable.looptune`.

See Also

`slTunable` | `slTunable.getBlockParam` | `slTunable.setBlockParam` | `slTunable.setBlockValue` | `slTunable.writeBlockValue`

slTunable.getIOTransfer

Purpose

Tunable model of closed-loop transfer function between specified inputs and outputs

Syntax

```
T = getIOTransfer(ST,input,output)
T = getIOTransfer(ST,input,output,openings)
T = getIOTransfer(ST)
```

Description

`T = getIOTransfer(ST,input,output)` returns the linearized closed-loop response of the control system described by the `slTunable` interface `ST`. The response `T` is a `genss` model of the response from `input` to `output`.

`T = getIOTransfer(ST,input,output,openings)` computes `T` with feedback loops opened at the locations specified by `openings`.

`T = getIOTransfer(ST)` returns the response from all inputs to all outputs. This syntax is equivalent to setting `inputs` and `outputs` to the union of the `IOs`, `Controls`, and `Measurements` properties of `ST`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

inputs

String or cell array of strings specifying the signals that are inputs to the closed-loop response. Each signal in `inputs` must be present in the `IOs`, `Controls`, `Measurements`, or `Switches` properties of `ST`. For signals in `ST.IOs`, only signals of type `input`, `sensitivity`, or `compsensitivity` can be used in `inputs`. The `getIOTransfer` command treats a signal in `Controls`, `Measurements`, or `Switches` as a disturbance signal entering at the specified location.

You can specify an abbreviated signal path in `inputs` provided the string matches the end of the full signal path and unambiguously identifies the signal.

Default: 0-by-1 cell array

outputs

String or cell array of strings specifying the signals that are outputs of the closed-loop response. Each signal in `outputs` must be present in the **`IOs`, `Controls`, or `Measurements`** properties of the `slTunable` interface `ST`. For signals in `ST.IOs`, only signals of type `output`, `sensitivity`, or `compsensitivity` can be used in `outputs`.

You can specify an abbreviated signal path in `outputs` provided the string matches the end of the full signal path and unambiguously identifies the signal.

Default: 0-by-1 cell array

openings

String or cell array of strings specifying the locations at which feedback loops are opened for the computation of `T`. Specify the locations as the names of signals listed in the `Controls`, `Measurements`, or `Switches` properties of `ST`. Loop openings listed in the `Openings` property of `ST` are always open.

You can specify an abbreviated signal path in `openings` provided the string matches the end of the full signal path and unambiguously identifies the signal.

Default: 0-by-1 cell array

Output Arguments

T

Tunable `genss` model of the response from inputs `input` to outputs `output`. If `input` and `output` are omitted, `T` is the response from all inputs to all outputs listed in the `IOs`, `Controls`, and `Measurements` properties of `ST`.

`T` captures the dependence of the response on the parameters of the tunable blocks in the `TunedBlocks` property of `ST`. The current values of the tunable parameters of `T` are equal to the current parameter values in `ST`. Therefore, you can use `getIOTransfer`

after tuning ST with `slTunable.systune` or `slTunable.looptune` to examine closed-loop responses of the tuned control system.

See Also

`slTunable` | `slTunable.loopview` | `slTunable.getLoopTransfer` | `genss`

Tutorials

- "Tuning Control Systems in Simulink"
- "Tuning Multi-Loop Control Systems"
- "Decoupling Controller for a Distillation Column"
- "Tuning of a Digital Motion Control System"
- "Fixed-Structure Autopilot for a Passenger Jet"

Purpose

Tunable model of open-loop transfer function at specified location

Syntax

```
L = getLoopTransfer(ST,location)
L = getLoopTransfer(ST,location,sign)
L = getLoopTransfer(ST,location,openings)
L = getLoopTransfer(ST,location,openings,sign)
```

Description

`L = getLoopTransfer(ST,location)` returns the linearized point-to-point open-loop transfer function of the control system described by the `slTunable` interface `ST`. The point-to-point open-loop transfer function `L` is the open-loop response obtained by injecting signals at `location` and measuring the return signals at the same point.

`L = getLoopTransfer(ST,location,sign)` specifies the feedback sign for computing the open-loop response. By default, `L` is the positive-feedback open-loop transfer function. Set `sign` to `-1` to compute the negative-feedback open-loop transfer function.

`L = getLoopTransfer(ST,location,openings)` computes `L` with additional loop openings at the locations specified by `openings`.

`L = getLoopTransfer(ST,location,openings,sign)` specifies both additional loop openings and the sign.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

location

String or cell array of strings specifying signal locations at which to compute a point-to-point open loop transfer function. Each signal in `location` must be present in the `Switches`, `Controls`, or `Measurements` properties of the `slTunable` interface `ST`. Use a cell array to specify multiple locations and compute a MIMO loop transfer.

You can specify an abbreviated signal path in `location` provided the string matches the end of the full signal path and unambiguously identifies the signal.

sign

Feedback sign. `sign` takes the following values:

- `+1` (default) — `getLoopTransfer` returns the positive-feedback open-loop transfer function, computed by injecting sine waves at the locations specified in `location` and measuring the return signals at the same point. The closed-loop transfer function is `feedback(L,1,+1)`.
- `-1` — `getLoopTransfer` returns the negative-feedback open-loop transfer function. The negative-feedback transfer function is –1 times the positive-feedback transfer function. The closed-loop transfer function is `feedback(L,1)`.

Set `sign` to `-1` if you intend to use the open-loop transfer function `L` in applications that assume the negative-feedback definition of `L`. Many classical design and analysis techniques use the negative-feedback convention. For example, the Nyquist or root locus design techniques usually assume that `L` is defined such that the closed-loop transfer function is `feedback(L,1)`.

Default: `+1`

openings

String or cell array of strings specifying additional locations at which feedback loops are opened for the computation of `L`. Specify the locations as the names of signals listed in the `Controls`, `Measurements`, or `Switches` properties of `ST`. Loop openings listed in the `Openings` property of `ST` are always open.

You can specify an abbreviated signal path in `openings` provided the string matches the end of the full signal path and unambiguously identifies the signal.

Default: `{0x1 cell}`

Output Arguments

`L`

Tunable `genss` model of the point-to-point response measured at `location`. If `location` is a cell array identifying multiple signals, `L` is a MIMO response.

`L` captures the dependence of the response on the parameters of the tunable blocks in the `TunedBlocks` property of `ST`. The current values of the tunable parameters of `L` are equal to the current parameter values in `L`. Therefore, you can use `getLoopTransfer` after tuning `ST` with `slTunable.systune` or `slTunable.looptune` to examine open-loop responses of the tuned control system.

See Also

`slTunable` | `slTunable.loopview` | `slTunable.getIOTransfer` | `genss`

Tutorials

- "Validating Results"
- "Tuning of a Digital Motion Control System"

slTunable.linearize

Purpose

Relinearize Simulink model to update `slTunable` interface

Syntax

```
linearize(STO)
blockvals = linearize(STO)
```

Description

`linearize(STO)` relinearizes the Simulink model described by the `slTunable` interface `STO`. Use `linearize` to update the stored linear representation of the Simulink model in `STO` after you make changes to the Simulink model. The update to `STO` includes:

- Updating the parametrization of the control architecture according to the blocks specified in the `TunedBlocks` property of `STO`.

  Note To synchronize tuned block current values with the Simulink model, use `slTunable.readBlockValue`.

- Evaluating the fixed (nontunable) portion of the control system.
- Computing a new linearization of the Simulink model, using the I/Os specified in the `IOs`, `Openings`, and `Switches` properties of `STO`, respectively. The new linearization uses the operating point and linearization options specified in the `OperatingPoint` and `LinearizeOptions` properties of `STO`, respectively.

`blockvals = linearize(STO)` returns a cell array containing the current values from the Simulink model of the tunable blocks stored in the `TunedBlocks` property of `STO`.

Tips

- You do not need to use `slTunable.linearize` every time you change properties of the `slTunable` interface. Commands that query the linearization stored in the `slTunable` interface automatically relinearize the Simulink model if you have changed any properties of `STO` since the last linearization. These commands include `slTunable.systune`,`slTunable.looptune`, `slTunable.getIOTransfer`, and `slTunable.getLoopTransfer`. You

can optionally use `slTunable.linearize` to force relinearization or control when relinearization occurs.

Input Arguments

STO

`slTunable` interface describing the control system to be tuned.

Output Arguments

blockvals

Current values of the tunable blocks of `STO`. `blockvals` is a cell array of state-space (`ss`) models. Each entry in `blockvals` is the current value of the corresponding entry in the `TunedBlocks` property of `STO`.

Examples

Update the linearization in an `slTunable` interface to use new linearization options.

Create the `slTunable` interface.

```
open_system('rct_cascade');
tunedblocks = {'C1','C2'};
STO = slTunable('rct_cascade',tunedblocks);
```

Create a linearization options set, and update the linearization stored in `STO`.

```
opts = linearizeOptions('BlockReduction','off');
STO.LinearizeOptions = opts;
linearize(STO);
```

The linearization uses the options stored in the `LinearizeOptions` property of `STO`.

Algorithms

`slTunable.linearize` linearizes your Simulink model using the algorithms described in “Exact Linearization Algorithm” on page 2-152.

See Also

slTunable | slTunable.systune | slTunable.looptune | slTunable.getLoopTransfer | slTunable.getIOTransfer | slTunable.readBlockValue | linearize | linearizeOptions

Purpose

Tune MIMO feedback loops in Simulink

Syntax

```
[ST,gam,info] = looptune(ST0,wc)
[ST,gam,info] = looptune(ST0,wc,Req1,Req2,...)
[ST,gam,info] = looptune(...,options)
```

Description

`[ST,gam,info] = looptune(ST0,wc)` tunes the Simulink control system described by the `slTunable` interface `ST0` to meet the following default requirements:

- 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`

`[ST,gam,info] = looptune(ST0,wc,Req1,Req2,...)` tunes the feedback loop to meet additional design requirements specified in one or more tuning goal objects `Req`. Omit `wc` to use the requirements specified in the `Req` objects instead of an explicit target crossover frequency and the default performance and robustness requirements.

`[ST,gam,info] = looptune(...,options)` specifies further options, including target gain margin, target phase margin, and computational options for the tuning algorithm.

Tips

- `slTunable.looptune` allows you to use the Robust Control Toolbox tuning command `looptune` on a control system described in a Simulink model. Using `slTunable` provides an interface between the Simulink model and the `looptune` command, allowing you to identify the tunable components of the system for `looptune` to tune. Using `slTunable.looptune` requires Robust Control Toolbox software.

Input Arguments

ST0

`slTunable` interface object representing the Simulink control system to tune. For information on how to create and configure the `slTunable` interface, see the `slTunable` reference page.

wc

Vector specifying the target crossover region `[wcmin,wcmax]`. The `looptune` command attempts to tune all loops in the control system so that the open-loop gain crosses 0 dB within the target crossover region.

A scalar `wc` specifies the target crossover region `[wc/2,2*wc]`.

Req

One or more `TuningGoal` objects specifying design requirements. Available requirement types include:

- `TuningGoal.Tracking` — Setpoint tracking requirement
- `TuningGoal.Gain` — Limit on transfer function gain
- `TuningGoal.LoopShape` — Target shape for open-loop response

For a complete list of the design requirements you can specify, see "Performance and Robustness Specifications for looptune".

options

Set of options for the `looptune` algorithm, specified using `looptuneOptions`. See `looptuneOptions` for information about the available options, including target gain margin and phase margin.

Output Arguments

ST

Tuned version of `ST0`.

Use `slTunable.loopview` to graphically validate the design. Use `slTunable.writeBlockValue` to apply the tuned parameter values to the Simulink model.

gam

Parameter indicating degree of success at meeting all tuning constraints. A value of `gam <= 1` indicates that all requirements are satisfied. A value of `gam >> 1` indicates failure to meet at least one requirement. Use `slTunable.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 requirements.

info

Data for validating tuning results, returned as a structure. To use the data in `info`, use the command `loopview(ST,info)` to visualize tuning constraints and validate the tuned design.

`info` contains the following tuning data:

Di,Do

Optimal input and output scalings, returned as state-space models. The scaled plant is given by `Do\G*Di`.

Specs

Design requirements used for tuning, returned as a vector of `TuningGoal` requirement objects.

Runs

Detailed information about each optimization run performed by `systune` when called by `looptune` for tuning (see "Algorithms" on page 7-310), returned as a data structure.

The contents of `Runs` are the `info` output of the call to `systune`. For information about the fields of `Runs`,

see the `info` output argument description on the `slTunable.systune` reference page.

Algorithms

`looptune` automatically converts target bandwidth, performance requirements, and additional design requirements into weighting functions that express the requirements as an H_∞ optimization problem. `looptune` then uses `slTunable.systune` to optimize tunable parameters to minimize the H_∞ norm. For more 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

`slTunable` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `slTunable.systune` | `slTunable.loopview` | `slTunable.getLoopTransfer` | `slTunable.getIOTransfer` | `slTunable.writeBlockValue` | `looptuneOptions`

Tutorials

- “Tuning Control Systems in Simulink”
- “Decoupling Controller for a Distillation Column”
- “Tuning of a Digital Motion Control System”
- “Tuning of a Two-Loop Autopilot”

How To

- “Structure of Control System for Tuning With looptune”
- “Set Up Your Control System for Tuning with looptune”
- “Performance and Robustness Specifications for looptune”

Purpose

Convert tuning setup for `looptune` to tuning setup for `systune`

Syntax

```
[STO,SoftReqs,HardReqs,
   sysopt] = looptuneSetup(looptuneInputs)
```

Description

`[STO,SoftReqs,HardReqs, sysopt] = looptuneSetup(looptuneInputs)` converts a tuning setup for `slTunable.looptune` into an equivalent tuning setup for `slTunable.systune`. The argument `looptuneInputs` is a sequence of input arguments for `slTunable.looptune` that specifies the tuning setup. For example,

```
[STO,SoftReqs,HardReqs,sysopt] = looptuneSetup(STO,wc,Req1,Req2,loopop
```

generates a set of arguments such that `looptune(STO,wc,Req1,Req2,loopopt)` and `systune(STO,SoftReqs,HardReqs,sysopt)` produce the same results.

Use this command to take advantage of additional flexibility that `slTunable.systune` offers relative to `slTunable.looptune`. For example, `slTunable.looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `slTunable.systune` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `slTunable.looptune` treats all tuning requirements as soft requirements, optimizing them but not requiring that any constraint be exactly met. Converting to `slTunable.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 `slTunable.looptune`.

Input Arguments

`looptuneInputs` - Control system and requirements configured for tuning with `slTunable.looptune`

valid `slTunable.looptune` input sequence

Plant, controller, and requirements configured for tuning with `slTunable.looptune`, specified as a valid `slTunable.looptune`

input sequence. For more information about the arguments in a valid `slTunable.looptune` input sequence, see the `slTunable.looptune` reference page.

Output Arguments

STO - Interface to Simulink model to tune

`slTunable` interface

Interface to Simulink model you are tuning, returned as an `slTunable` interface. `STO` is identical to the `slTunable` interface you use as input to `looptuneSetup`.

SoftReqs - Soft tuning requirements

vector of `TuningGoal` requirement objects

Soft tuning requirements for tuning with `slTunable.systune`, specified as a vector of `TuningGoal` requirement objects.

`slTunable.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, `slTunable.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 `slTunable.systune`, specified as a vector of `TuningGoal` requirement objects.

Because `slTunable.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

(default) | `systuneOptions` options set

Algorithm options for `slTunable.systune` tuning, specified 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 `systtuneOptions` set.

Examples

Convert looptune Problem into systune Problem

Convert a set of `slTunable.looptune` inputs for tuning a Simulink model into an equivalent set of inputs for `slTunable.systune`.

Suppose you have created and configured an `slTunable` interface, `STO`, for tuning with `slTunable.looptune`. Suppose also that you used `slTunable.looptune` to tune the feedback loop defined in `STO` to within a bandwidth of `wc = [wmin,wmax]`. Convert these variables into a form that allows you to use `slTunable.systune` for further tuning.

```
[STO,SoftReqs,HardReqs,sysopt] = looptuneSetup(STO,wc);
```

The command returns the closed-loop system and tuning requirements for the equivalent `slTunable.systune` command, `systune(STO,SoftReqs,HardReqs,sysopt)`. The arrays `SoftReqs` and `HardReqs` contain the tuning requirements implicitly imposed by `slTunable.looptune`. These requirements enforce the target bandwidth and default stability margins of `slTunable.looptune`.

If you used additional tuning requirements when tuning the system with `slTunable.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 `slTunable.systune` command.

```
[STO,SoftReqs,HardReqs,sysopt] = looptuneSetup(STO,wc,Req1,Req2);
```

The resulting arguments allow you to construct the equivalent tuning problem for `slTunable.systune`.

Convert Distillation Column Problem for Tuning With *systune*

Set up the control system of the Simulink model `rct_distillation` for tuning with `slTunable.looptune`. Then convert the setup to a `slTunable.systune` problem, and examine the resulting arguments. The results reflect the tuning requirements implicitly enforced when tuning with `looptune`.

Create an `slTunable` interface to the Simulink model. Configure the interface for tuning with `slTunable.looptune` by specifying control and measurement signals that define the separation between the plant and the controller. Also specify analysis I/Os for imposing tuning requirements.

```
open_system('rct_distillation')

% Tuned blocks (specified by their name in the model)
ST0 = slTunable('rct_distillation',{'PI_L','PI_V','DM'});

% Control and measurement signals
ST0.addControl({'L','V'})
ST0.addMeasurement('y')

% Analysis I/Os
ST0.addIO('r','input')          % reference inputs
ST0.addIO({'dL','dV'},'input')  % disturbances
```

This system is now ready for tuning with `slTunable.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];
TR = TuningGoal.Tracking('r','y',15);
MaxGain = frd([0.05 5 5],[0.001 0.1 10],'TimeUnit','min');
DR = TuningGoal.MaxGain({'dL','dV'},'y',MaxGain);
```

```
[ST,gam,Info] = STO.looptune(wc,TR,DR);
```

```
Final: Peak gain = 1.03, Iterations = 102
```

`slTunable.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 `TR` and `DR` as hard constraints, and add other requirements as soft requirements.

Convert the `slTunable.looptune` input arguments to a set of input arguments for `slTunable.systune`.

```
[STO,SoftReqs,HardReqs,sysopt] = looptuneSetup(STO,wc,TR,DR);
```

This command returns a set of arguments you can feed to `slTunable.systune` for equivalent results to tuning with `slTunable.looptune`. In other words, the following command is equivalent to the `slTunable.looptune` command.

```
[ST,fsoft,ghard,info] = systune(STO,SoftReqs,HardReqs,sysopt);
```

```
Final: Peak gain = 1.03, Iterations = 102
```

Examine the tuning requirements returned by `looptuneSetup`. When tuning this control system with `slTunable.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 LoopGeneric (LoopShape, Tracking, Rejection, ...)
```

```
    Models
    Openings
    Name
```

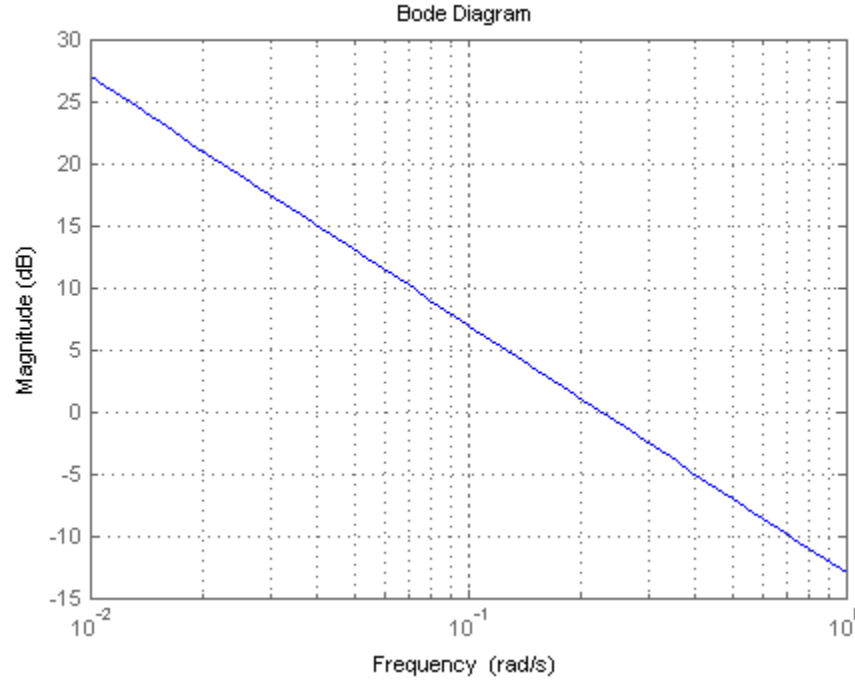
For example, examine the first entry in `SoftReqs`.

```
SoftReqs(1)
```

```
ans =

  LoopShape with properties:

       Location: {2x1 cell}
       LoopGain: [1x1 zpk]
       CrossTol: 0.3495
    LoopScaling: 'on'
          Focus: [0 Inf]
         Models: NaN
       Openings: {0x1 cell}
           Name: 'Open loop CG'
```

`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 `slTunable.systune`.

`slTunable.looptune` also tunes to default stability margins that you can change using `looptuneOptions`. For `slTunable.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:

       Location: {2x1 cell}
    GainMargin: 7.6000
   PhaseMargin: 45
         Focus: [0 Inf]
        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`.

See Also

`slTunable` | `slTunable.looptune` | `slTunable.systune` | `looptuneOptions` | `systuneOptions` | `looptuneSetup`

Purpose

Graphically analyze results of control system tuning

Syntax

```
loopview(ST)
loopview(ST,info)
```

Description

`loopview(ST)` plots characteristics of the control system described by the `slTunable` interface `ST`.

Use `loopview` to analyze the performance of a tuned control system you obtain using `slTunable.looptune`.

`loopview` plots the singular values of:

- Open-loop frequency responses `G*C` and `C*G`. The controller `C` and plant `G` are the linearized portions of the Simulink model defined by the `Controls` and `Measurements` properties of `ST`.
- Sensitivity function `S = inv(1+G*C)` and complementary sensitivity `T = 1-S`
- Normalized multi-loop disk margins (see `loopmargin`). `loopview` plots the:
  - Maximum stability margin
  - Tuned (optimized) stability margin
  - Actual (achieved) stability margin

  Use the stability margin plot to verify that the tuned control system has the desired stability margin. The normalized margin value should be less than 1 at all frequencies.

For more information about singular values, see `sigma`.

`loopview(ST,info)` uses the `info` structure returned by `slTunable.looptune`. This syntax also plots the target and tuned values of tuning constraints imposed on the system. Additional plots include:

- Maximum allowed singular values for the sensitivity functions `S` and `T`. The curve marked `S/T Max` shows the maximum allowed `S` on the low-frequency side of the plot, and the maximum allowed `T` on the high-frequency side. These curves are the constraints that `looptune` imposes on `S` and `T` to enforce the target crossover range `wc`.
- Target and tuned values of constraints imposed by any tuning goal requirements you used with `looptune`.

Use `loopview` with the `info` structure to assist in troubleshooting when tuning fails to meet all requirements.

Input Arguments

`ST`

`slTunable` interface describing the Simulink model to analyze.

`info`

`info` structure returned by `slTunable.looptune` during control system tuning.

Alternatives

For analyzing Control System Toolbox models tuned with `looptune`, use `loopview`.

See Also

`looptune` | `loopview` | `slTunable.looptune`

Tutorials

- "Decoupling Controller for a Distillation Column"
- "Tuning of a Two-Loop Autopilot"

Purpose

Update tuned block values from Simulink model

Syntax

```
readBlockValue(ST)
readBlockValue(ST,block)
```

Description

`readBlockValue(ST)` updates the current value of the block parametrization of the interface `ST` to match the blocks values in the corresponding Simulink model. This command only updates Simulink blocks parametrized by Control Design Blocks. For information about block parametrization, see `slTunable.getBlockParam`.

`readBlockValue(ST,block)` updates parameters only for the block or blocks specified by `block`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String or cell array of strings identifying blocks in the `TunedBlocks` property of `ST` to update with values read from the Simulink model. You can specify a block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

See Also

`slTunable | slTunable.writeBlockValue | slTunable.getBlockParam | slTunable.getBlockValue | slTunable.setBlockValue | slTunable.showBlockValue`

slTunable.setBlockParam

Purpose

Specify parametrization for specified tuned block

Syntax

```
setBlockParam(ST,block,TC)
```

Description

`setBlockParam(ST,block,TC)` assigns the tunable model `TC` as the parametrization of the tunable block, `block`, of the `slTunable` interface `ST`. Using `setBlockParam` overrides the default parametrization for the specified block. Doing so defines how the block should be parametrized and which parameters of the block are tuned by tuning commands involving `ST`.

Tips

- The `slTunable` interface stores a parametrization of each blocks listed in the `TunedBlocks` property of the interface. This parametrization specifies which parameters are tuned by commands such as `slTunable.looptune`. Use `setBlockParam` to override the default block parametrization or specify how to parametrize composite Simulink blocks such as subsystems.
- In some cases, `slTunable.writeBlockValue` cannot write tuned parameter values from a custom parametrization to the Simulink model. `slTunable.writeBlockValue` can only write tuned parameter values when the value is compatible with the default parametrization. For example, you might override the default parametrization of a static `Gain` block (`ltiblock.gain`) with a transfer function, (`ltiblock.tf`). In that case, `slTunable.writeBlockValue` errors unless the current value of the custom parametrization transfer function is a static gain.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

TC

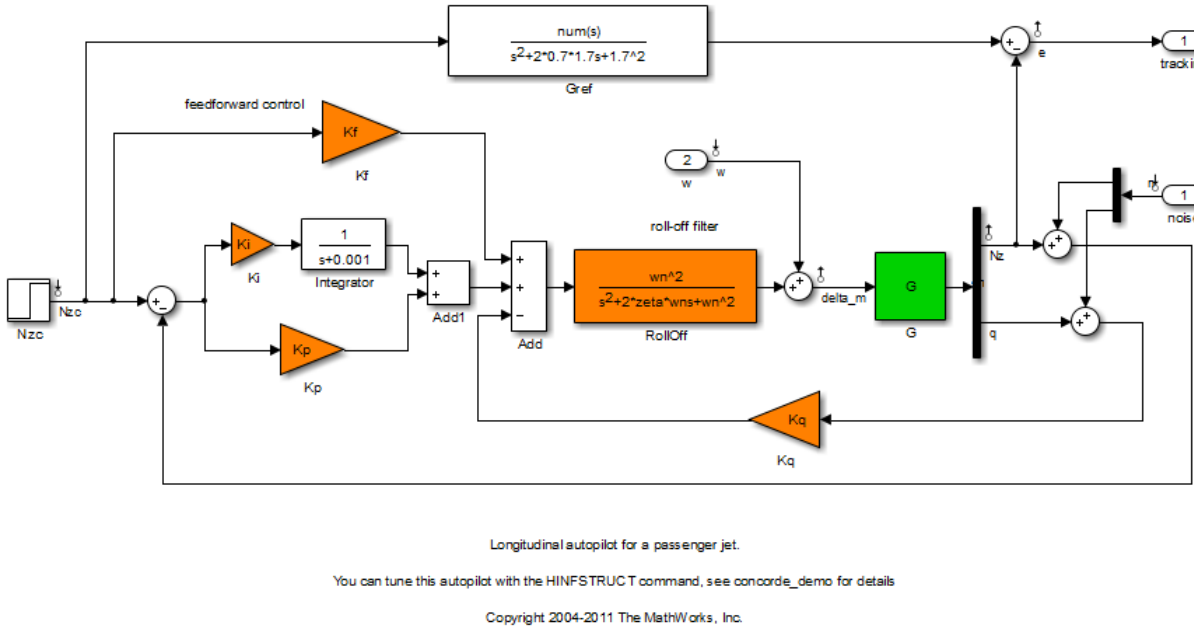
Parametrization to assign to `block`. Express the parametrization as a tunable Control Design Block or `genss` model.

Examples

Create an `slTunable` interface describing the Simulink model `rct_concorde` and specify the parametrization of a tunable block.

Open the Simulink model.

```
open_system('rct_concorde')
```



Create an `slTunable` interface for the model with the tunable blocks shown in orange.

```
ST = slTunable('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

Examine the default parametrization of the filter block `Rolloff`.

```
getBlockParam(ST,'RollOff')
```

```
Parametric continuous-time transfer function "rct_concorde_RollOff".
This transfer function has 0 zeros, 2 poles, and 3 tunable parameters.

Use "tf(M)" to see the current value and "get(M)" to see all properties.
```

The default parametrization is a generic second-order transfer function in which all coefficients are free to vary independently. The desired parametrization is a second-order filter where the coefficients are constrained by the relationship

$$F_{ro}(s) = \frac{\omega_n^2}{s^2 + 2\zeta_n\omega_n s + \omega_n^2}.$$

Create a `genss` model representing the desired parametrization, and set it as the parametrization of the `'RollOff'` block in `ST`.

```
wn = realp('wn', 3);
zeta = realp('zeta',0.8);
Fro = tf(wn^2,[1 2*zeta*wn wn^2]);

setBlockParam(ST,'RollOff',Fro)
```

Tuning commands such as `slTunable.systune` now use this constrained parametrization of the `RollOff` block of `ST`.

This custom parametrization is compatible with the default parametrization of the Simulink block. Therefore, you can use `slTunable.writeBlockValue` to write the tuned values back to the block.

See Also

`slTunable` | `slTunable.getBlockValue` | `slTunable.getBlockParam` | `slTunable.writeBlockValue` | `slTunable.systune` | `genss`

How To

- “Supported Blocks for Tuning in Simulink”

Purpose

Set rate conversion method of tunable block

Syntax

```
setBlockRateConversion(ST,block,method)
setBlockRateConversion(ST,block,method,pwfreq)
```

Description

`setBlockRateConversion(ST,block,method)` sets the rate conversion method currently associated with a tunable block in an `slTunable` interface.

`setBlockRateConversion(ST,block,method,pwfreq)` also sets the prewarp frequency for the `'tustin'` method.

Tips

- Use `setBlockRateConversion` to query the rate conversion method that the `slTunable` interface uses when converting the sampling time of the block parametrization.

  For example, you might use the `slTunable` interface to tune a mixed-rate Simulink model having a discrete-time controller and a continuous time plant. Tuning commands such as `looptune` or `systune` tune the model in continuous time. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, `writeBlockValue` automatically discretizes the tuned parametrization. It uses the block rate conversion method associated with the block for this discretization.

  For information about choosing discretization methods, see "Continuous-Discrete Conversion Methods".

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single tunable block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, if the string matches the end of the full block path and unambiguously identifies the block.

method

Discretization method associated with the tunable block, specified as one of the following strings:

- `'zoh'` — Zero-order hold method
- `'tustin'` — Tustin method

pwfreq

Prewarp frequency for the Tustin method, specified as a positive scalar.

For Tustin without prewarp, use `pwfreq = 0`. For zero-order hold, `pwfreq` is ignored.

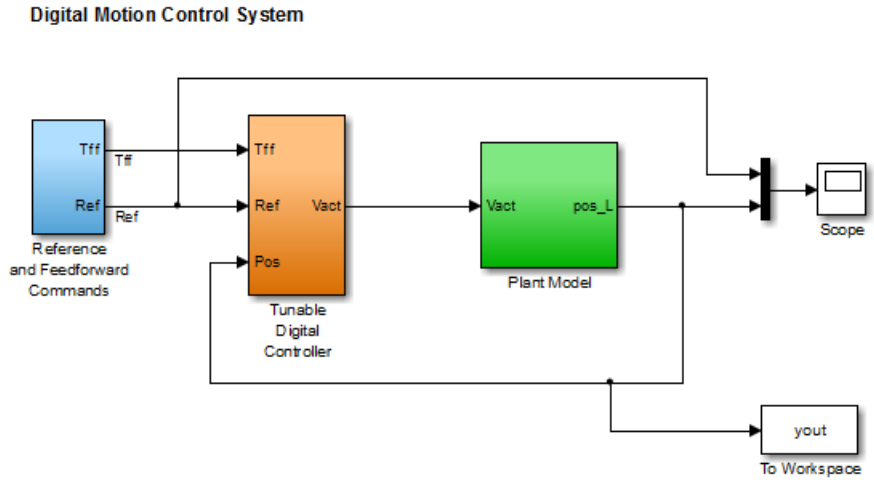
Default: 0

Examples

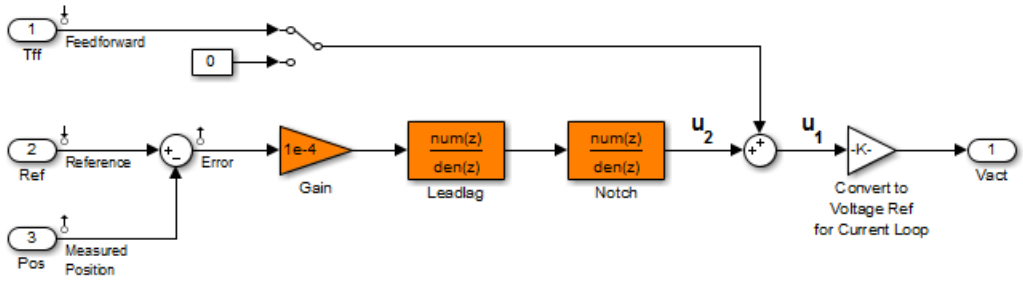
Create an `slTunable` interface describing the Simulink model `rct_dmcNotch`. Query and change the rate conversion method associated with a tunable block in the model.

Open the Simulink model.

```
open_system('rct_dmcNotch')
```



Double-click the `Tunable Digital Controller` subsystem to see the tunable blocks.



Create an `slTunable` interface for the model with the tunable blocks shown in orange.

```
ST = slTunable('rct_dmcNotch',{'Gain','Leadlag','Notch'});
```

The `LeadLag` and `Notch` blocks are discrete-time blocks. You might tune this control system in continuous time (for example, using `looptune` or

systune) and write the tuned parameters back to the Simulink model. If you do so, slTunable.writeBlockValue automatically discretizes the resulting tuned blocks using the discretization method associated with the Simulink block. Query the discretization method associated with the Notch block.

```
method = getBlockRateConversion(ST,'Notch')
```

```
method =

zoh
```

The result shows that the zero-order-hold (zoh) method is currently associated with the Notch block. This block represents a notch filter. Therefore, you may get better discretization results using the Tustin discretization method with a prewarp at the notch frequency. Assume that the tuned notch frequency is 340 rad/s.

```
wn = 340;
setBlockRateConversion(ST,'Notch','tustin',wn)
```

This command sets the rate conversion method of the Notch block to the Tustin method with a prewarp at the notch frequency. slTunable.writeBlockValue now discretizes the tuned block using this method.

For a more detailed treatment of this example, see "Tuning of a Digital Motion Control System".

See Also

slTunable | slTunable.writeBlockValue | slTunable.getBlockRateConversion

Tutorials

- "Tuning of a Digital Motion Control System"

How To

- "Continuous-Discrete Conversion Methods"

Purpose

Set current value of parametrization for specified block

Syntax

```
setBlockValue(ST,block,val)
setBlockValue(ST,M)
```

Description

`setBlockValue(ST,block,val)` sets to `val` the current value of the parametrization of the Simulink block `block` in the `slTunable` interface `ST`. The parametrization of `block` must be a tunable Control Design Block.

`setBlockValue(ST,M)` updates the current values of the tunable parameters in `ST` to match the corresponding values of tunable parameters in the Generalized LTI Model `M`. Only parameters common to `ST` and `M` are updated. Use this syntax to update `ST` after obtaining `M` by tuning with commands such as `systune`, `looptune`, or `hinsfstruct`.

Tips

- The `slTunable` interface automatically associates a parametric model with each Simulink block listed in the `TunedBlocks` property of `ST`. This parametrization expresses each tunable block as a Control Design Block or a tunable `genss` model. The parametrization specifies which parameters are tuned by commands such as `slTunable.looptune`. Use `slTunable.getBlockValue` to access the current value of this parametrization.
- You can use `setBlockValue` to initialize the tunable parameters of blocks parametrized by Control Design Blocks before tuning `ST` with a tuning command such as `slTunable.systune` or `slTunable.looptune`.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

String identifying a single block in the `TunedBlocks` property of `ST`. You can specify the block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

val

New initial or current value for the block parametrization. The value `val` can be any value that is compatible with the parametrization of `block` without changing the size, type, or sampling time the parametrization of `block`. For example, if the parametrization of `block` is a `ltiblock.pid` model, valid types for `val` include `ltiblock.pid`, a numeric `pid` controller model, or a numeric `tf` model that represents a PID controller. `setBlockValue` uses the parameter values of `val` to set the current value of the parametrization of `block`.

`setBlockValue` changes the current value of the block parametrization but does not change the parametrization itself. To change the parametrization (for example, to change the block parametrization from `ltiblock.pid` to a three-pole `ltiblock.tf`), use `slTunable.setBlockParam`.

M

Generalized LTI Model that has at least some parameters in common with `ST`.

See Also

`slTunable` | `slTunable.getBlockParam` | `slTunable.setBlockParam` | `slTunable.getBlockValue` | `slTunable.writeBlockValue`

Tutorials

- "Fixed-Structure Autopilot for a Passenger Jet"

Purpose

Display current value of block parametrizations

Syntax

```
showBlockValue(ST)
```

Description

`showBlockValue(ST)` is the same as `showTunable(ST)` for `slTunable` interface objects.

See Also

`slTunable.showTunable`

slTunable.showTunable

Purpose

Display current value of block parametrizations

Syntax

```
showTunable(ST)
```

Description

`showTunable(ST)` displays the current values of the parametric models associate with each tunable block in the `slTunable` interface, `ST`.

Tips

- The `slTunable` interface automatically associates a parametric model with each Simulink block listed in the `TunedBlocks` property of `ST`. This parametrization expresses each tunable block as a Control Design Block or a tunable `genss` model. The `showBlockValue` command displays the current values of these tunable blocks.

Input Arguments

ST

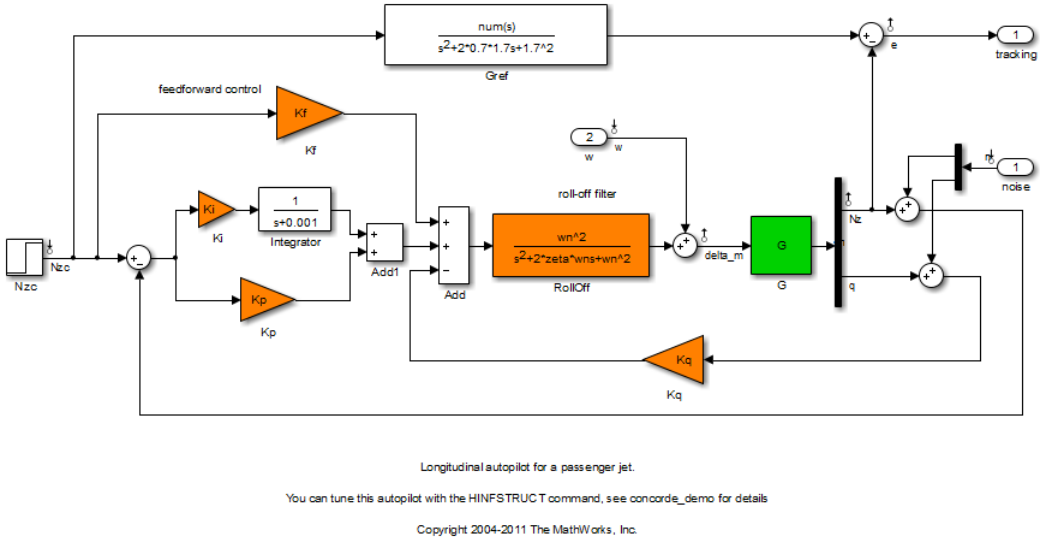
`slTunable` interface describing a Simulink model.

Examples

Create an `slTunable` interface describing the Simulink model `rct_concorde`. Display the values of the parametrizations of the tunable blocks.

Open the Simulink model.

```
open_system('rct_concorde')
```



Create an `slTunable` interface for the model with the tunable blocks shown in orange.

```
ST = slTunable('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

Examine the default parametrization values of the blocks.

```
showTunable(ST)
```

```
Block "rct_concorde/Ki" =

d =
       u1
   y1   0

Static gain.
-----------------------------------

Block "rct_concorde/Kp" =
```

```
d =
       u1
   y1   0

Static gain.
-----------------------------------

Block "rct_concorde/Kq" =

d =
       u1
   y1   0

Static gain.
-----------------------------------

Block "rct_concorde/Kf" =

d =
       u1
   y1   0

Static gain.
-----------------------------------

Block "rct_concorde/RollOff" =

Transfer function:
       9
---------------
s^2 + 4.8 s + 9
```

The default parametrization for the four gain blocks is a static gain, with a current value of 0. The default parametrization for the filter `RollOff` is a second-order transfer function.

Use `slTunable.setBlockValue` to change the current values of the parameters. Use `slTunable.setBlockParam` to change the parametrization.

See Also

`slTunable` | `slTunable.writeBlockValue` | `slTunable.getBlockValue` | `slTunable.setBlockValue` | `slTunable.readBlockValue`

slTunable.systune

Purpose

Tune fixed-structure control system parameters in Simulink

Syntax

```
[ST,fSoft] = systune(STO,SoftReqs)
[ST,fSoft,gHard] = systune(STO,SoftReqs,HardReqs)
[ST,fSoft,gHard] = systune(STO,SoftReqs,HardReqs,options)
[ST,fSoft,gHard,info] = systune( ___ )
```

Description

`[ST,fSoft] = systune(STO,SoftReqs)` tunes the free parameters of the Simulink control system described by the `slTunable` interface, `STO`. The `systune` command tunes the parameters to best meet the soft tuning requirements `SoftReqs`, and returns the best achieved soft constraint values as `fSoft`.

`[ST,fSoft,gHard] = systune(STO,SoftReqs,HardReqs)` tunes the control system to best meet the soft tuning requirements subject to satisfying the hard tuning requirements (constraints). It returns the best achieved values for the soft and hard constraints.

`[ST,fSoft,gHard] = systune(STO,SoftReqs,HardReqs,options)` specifies options for the optimization.

`[ST,fSoft,gHard,info] = systune( ___ )` also returns detailed information about each optimization run.

Tips

- `slTunable.systune` allows you to use the Robust Control Toolbox tuning command, `systune`, on a control system described in a Simulink model. Using `slTunable` provides an interface between the Simulink model and the `systune` command, allowing you to identify the tunable components of the system for `systune` to tune. Using `slTunable.systune` requires Robust Control Toolbox software.

Input Arguments

STO

`slTunable` interface object representing the Simulink control system to tune. For information on how to create and configure the `slTunable` interface, see the `slTunable` reference page.

SoftReqs

Soft tuning requirements (objectives) for tuning the control system, specified as a vector of `TuningGoal` requirement objects. These requirement objects can include `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.Margins`. (See "Specifying Design Requirements for systune" for a complete list.)

`systune` tunes the tunable parameters of the control system to minimize the soft tuning requirements, subject to satisfying the hard tuning requirements (if any).

HardReqs

Hard tuning requirements (constraints) for tuning the control system, specified as a vector of `TuningGoal` requirement objects. These requirement objects can include `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.Margins`. (See "Specifying Design Requirements for systune" for a complete list.)

`systune` converts each hard tuning requirement to a normalized scalar value. `systune` then optimizes the free parameters minimize those normalized values. A hard requirement is satisfied if the normalized value is less than 1.

Default: `[]`

options

Options for the tuning algorithm, specified as an options set you create with `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

Output Arguments

ST

Tuned version of `ST0`.

Use `viewSpec` to graphically validate the design. Use `slTunable.writeBlockValue` to apply the tuned parameter values to the Simulink model.

fSoft

Best achieved soft constraint values, returned as a vector. `systune` converts the soft requirements 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 7-341.) `fSoft` contains the best achieved value for each of the soft constraints. These values appear in `fSoft` in the same order that the constraints are specified in `SoftReqs`. `fSoft` values are only meaningful when the hard constraints are satisfied.

gHard

Best achieved hard constraint values, returned as a vector. `systune` converts the hard requirements 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 7-341.) `gHard` contains the best achieved value for each of the hard constraints. These values appear in `gHard` in the same order that the constraints are specified in `HardReqs`. If all values are less than 1, then the hard constraints are satisfied.

info

Detailed information about each optimization run, returned as a data structure. In addition to examining detailed results of the optimization, you can use `info` as an input to `viewSpec` when validating a tuned MIMO system. `info` contains scaling data that `viewSpec` needs for correct evaluation of MIMO open-loop requirements, such as loop shapes and stability margins.

The fields of `info` are:

Run

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, returned as a scalar.

fBest

Best overall soft constraint value, returned as a scalar. `systune` converts the soft requirements 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 7-341.) `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, returned as a scalar. `systune` converts the hard requirements 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 7-341.) `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, 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 7-341.) `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 `SoftReqs`.

gHard

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 7-341.) `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 `HardReqs`.

MinDecay

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 in the tuned control system `CL`, returned as a structure. You can also use `getBlockValue` or `showBlockValue` to access the tuned parameter values.

LoopScaling

Optimal diagonal scaling for evaluating MIMO tuning requirements, returned as a state-space model.

When applied to multiloop control systems, `TuningGoal.LoopShape` and `TuningGoal.Margins` requirements 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`.

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 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.

`systune` returns a control system `ST` 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.

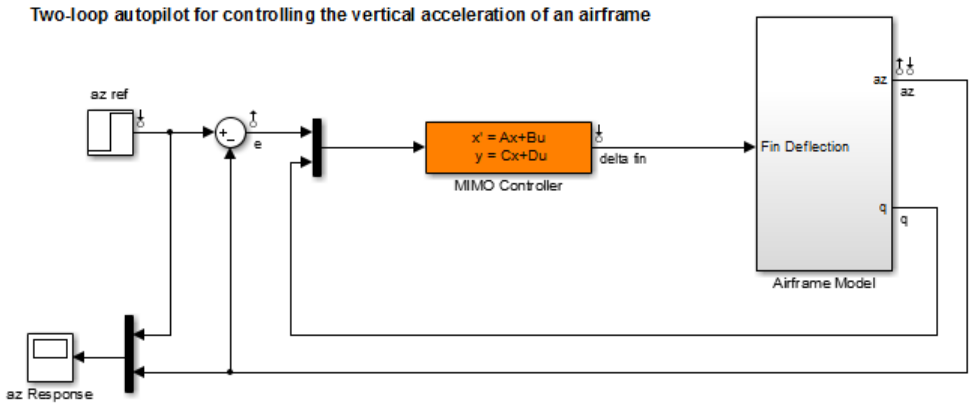
For information about the optimization algorithms, see [1].

`systune` 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.

Examples

Tune Control System to Soft Constraints

Tune the following control system to soft requirements for tracking, roll off, stability margin, and disturbance rejection.



Create and configure an `slTunable` interface to the model.

```
open_system('rct_airframe2')
STO = slTunable('rct_airframe2','MIMO Controller');

addControl(STO,'delta fin');
```

These commands create an `slTunable` interface to the model and specify the `MIMO Controller` block as the tunable portion of the control system.

The model already has linearization input points on the signals `az ref`, `delta fin`, and `az`. These signals are therefore available for specifying as inputs for tuning requirements. Similarly, you can use the linearization output points `e` and `az` as outputs for tuning requirements. The `addControl` command adds `delta fin` to the `Controls` signals of the `slTunable` interface. Doing so makes `delta fin` available as both input and output for specifying tuning requirements.

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);
```

```
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);
```

`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 requirements.

```
Options = systuneOptions('RandomStart',3);
[ST1,fSoft,~,Info] = systune(ST0,[Req1,Req2,Req3,Req4],Options);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 55
Final: Soft = 1.5, Hard = -Inf, Iterations = 75
Final: Soft = 1.13, Hard = -Inf, Iterations = 92
```

`ST1` is a tuned version of the `slTunable` interface to the control system that contains the tuned values of the tunable parameters of the MIMO controller in the model.

The `RandomStart` option causes `systune` to perform three independent optimization runs beginning at different random values of the tunable parameters. The displayed value, `Soft`, is the largest normalized soft constraint value for each optimization run.

Examine the best achieved values of the soft constraints.

```
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 requirements for at least some frequencies. Use `viewSpec` to visualize

the tuned control system performance against the requirements and to determine whether the violations are acceptable.

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

`slTunable` | `systune` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins` | `TuningGoal.WeightedGain` | `TuningGoal.Variance` | `TuningGoal.WeightedVariance` | `TuningGoal.LoopShape` | `TuningGoal.Poles` | `TuningGoal.StableController`

How To

- “Specifying Design Requirements for systune”
- “Tuning Control Systems in Simulink”
- “Control of a Linear Electric Actuator”

Purpose

Update block values in Simulink model

Syntax

```
writeBlockValue(ST)
writeBlockValue(ST,M)
writeBlockValue( ___ ,block)
```

Description

`writeBlockValue(ST)` writes tuned parameter values from an `slTunable` interface to the Simulink model that `ST` describes. Use this command, for example, to validate parameters of a control system that you tuned using `slTunable.systune` or `slTunable.looptune`.

`writeBlockValue(ST,M)` writes tuned parameter values from a generalized model, `M`, to the Simulink model described by an `slTunable` interface, `ST`. This syntax also updates the current values of the tunable parameters in `ST` to match their values in `M`. Only parameters common to both `ST` and `M` are updated. This syntax is equivalent to:

```
setBlockValue(ST,M);
writeBlockValue(ST);
```

Use this syntax when `M` is a model that you have tuned using a function like `systune`, `looptune`, or `hinfstruct`. For example, `M` might be a generalized model that you obtained by linearizing your Simulink model, and then tuned to meet some design requirements.

`writeBlockValue( ___ ,block)` updates only the specified block or blocks specified. You can use the `block` argument with any of the previous syntaxes.

Tips

- `writeBlockValue` skips blocks that:
  - Have no default parametrization
  - Have a parametrization specified with `slTunable.setBlockParam` that is not compatible with the default parametrization. For example, you might specify the parametrization of a susbsystem or S-Function block in your model. In that case, `slTunable.writeBlockValue` cannot write the tuned parameter values to the model.

Input Arguments

ST

`slTunable` interface describing a Simulink model.

block

Tunable blocks of `ST` to write back to the Simulink model, specified as a string or cell array of strings that are listed in the `TunedBlocks` property of `ST`. You can specify a block with an abbreviated block name, provided that the string matches the end of the full block path and unambiguously identifies the block.

M

Control system with tuned parameters to write back to the Simulink model, specified as a generalized state-space (`genss`) model. 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` might be a generalized model that you obtained by linearizing your Simulink model, and then tuned to meet some design requirements.

See Also

`slTunable` | `slTunable.readBlockValue` | `slTunable.getBlockValue` | `slTunable.setBlockValue` | `slTunable.showBlockValue`

Tutorials

- "Tuning of a Digital Motion Control System"
- "Control of a Linear Electric Actuator"

How To

- "Supported Blocks for Tuning in Simulink"

Purpose

Update operating point object with structural changes in model

Syntax

```
update(op)
```

Alternatives

As an alternative to the `update` function, update operating point objects using the **Sync with Model** button in the Simulink Control Design GUI.

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:

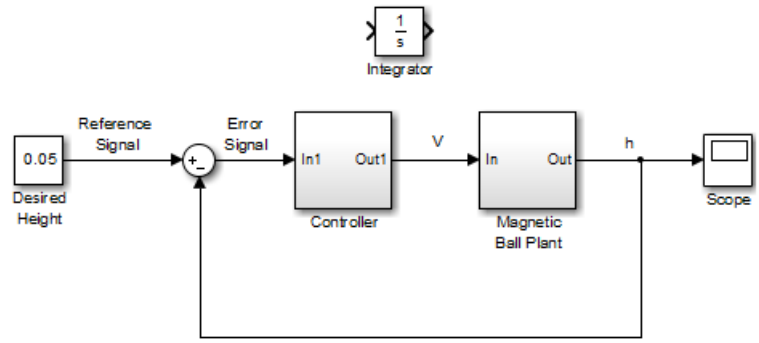
```
op=operpoint('magball')
```

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

See Also

operpoint | operspec

8

Blocks — Alphabetical List

Bode Plot

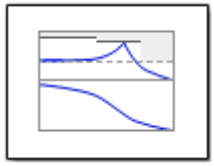
Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 8-315. • "Use exact delays" on page 8-317. • "Linear system sample time" on page 8-318. • "Sample time rate conversion method" on page 8-320.

Task		Parameters
		• "Prewarp frequency (rad/s)" on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 8-324. • "Use bus signal names" on page 8-326.
Plot the linear system.		**Show Plot**
(Optional) Specify bounds on magnitude of the linear system for assertion.		In **Bounds** tab: • "Include upper magnitude bound in assertion" on page 8-30. • "Include lower magnitude bound in assertion" on page 8-36.
Specify assertion options (only when you specify bounds on the linear system).		In **Assertion** tab: • "Enable assertion" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.
Save linear system to MATLAB workspace.		"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.		"Show plot on block open" on page 8-351.

Linearization inputs/outputs

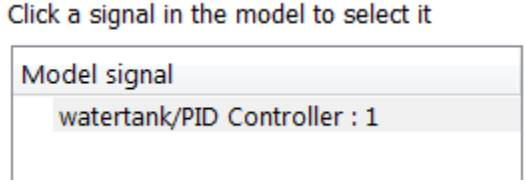
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** area and a new button.

2. Select a signal in the Simulink Editor.

   The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select an individual element.

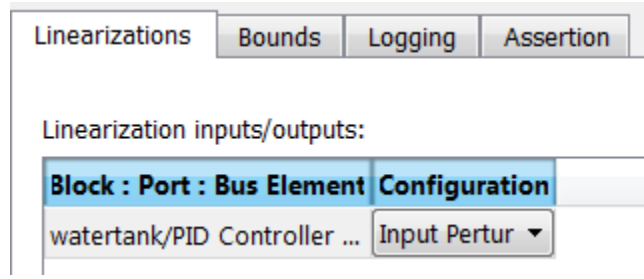
Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression.

To modify the filtering options, click .

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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".

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

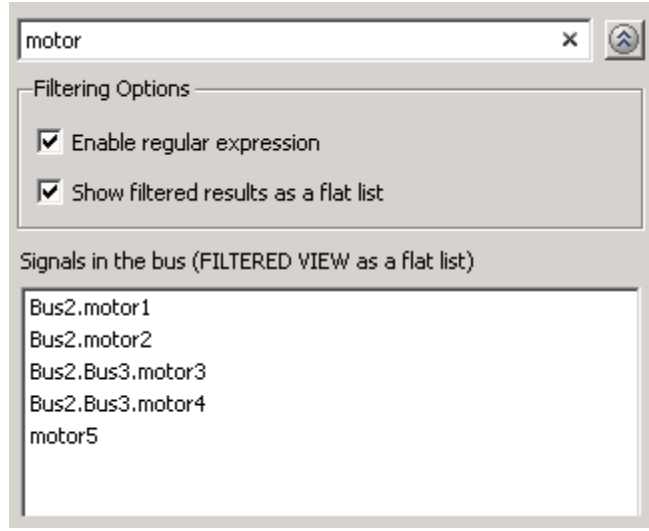
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

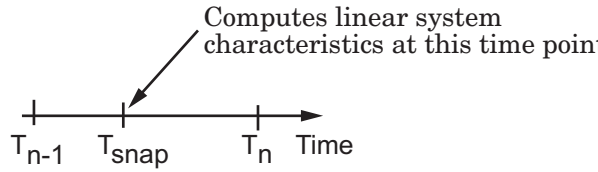
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

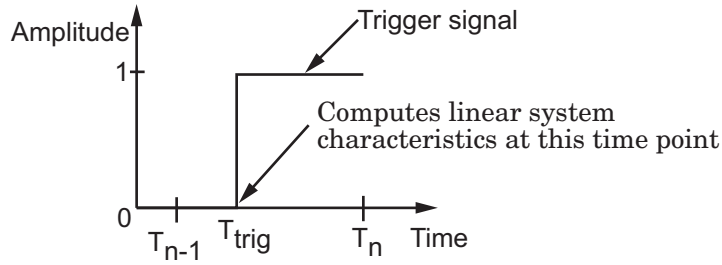
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`

Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include upper magnitude bound in assertion

Check that the Bode response satisfies upper magnitude bounds, specified in **Frequencies (rad/sec)** and **Magnitude (dB)**, during simulation. The software displays a warning if the magnitude violates the upper bounds.

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

You can specify multiple upper 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.

Settings

Default:

- Off for Bode Plot block.
- On for Check Bode Characteristics block.

On

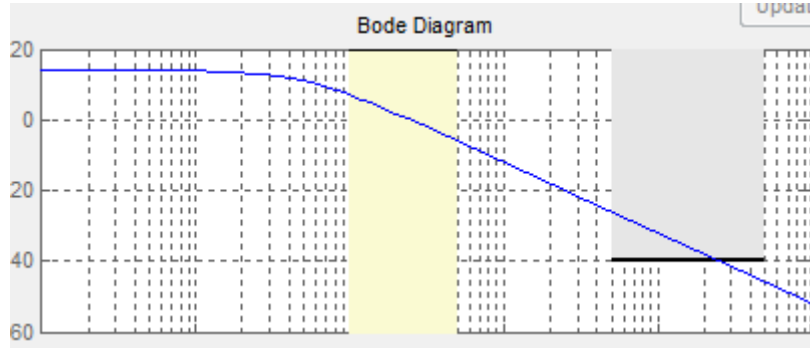
Check that the magnitude satisfies the specified upper bounds, during simulation.

Off

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

Tips

- 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 but want to include only the lower bounds for assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableUpperBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Frequencies (rad/sec)

Frequencies for one or more upper magnitude bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)**.

Settings

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

Tips

- To assert that magnitudes that correspond to the frequencies are satisfied, select both **Include upper magnitude bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `UpperBoundFrequencies`
Type: string
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

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Magnitudes (dB)

Magnitude values for one or more upper magnitude bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)**.

Settings

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

Tips

- To assert that magnitude bounds are satisfied, select both **Include upper magnitude bound in assertion** and **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 `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.

Command-Line Information

Parameter: `UpperBoundMagnitudes`
Type: string
Value: `[]` | `[-20 -20]` | `finite numbers` | `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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include lower magnitude bound in assertion

Check that the Bode response satisfies lower magnitude bounds, specified in **Frequencies (rad/sec)** and **Magnitude (dB)**, during simulation. The software displays a warning if the magnitude violates the lower bounds.

This parameter is used for assertion only if **Enable assertion** 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.

Settings

Default:

- Off for Bode Plot block.
- On for Check Bode Characteristics block

On

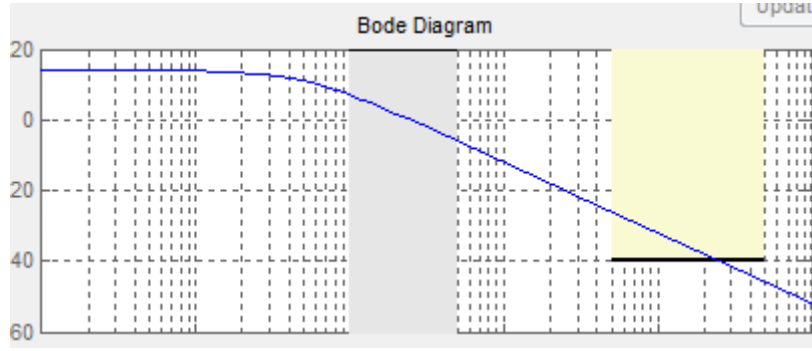
Check that the magnitude satisfies the specified lower bounds during simulation.

Off

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

Tips

- 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**.

Command-Line Information

Parameter: `EnableLowerBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Frequencies (rad/sec)

Frequencies for one or more lower magnitude bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)**.

Settings

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

Tips

- To assert that magnitude bounds that correspond to the frequencies are satisfied, select both **Include lower magnitude bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `LowerBoundFrequencies`
Type: string
Value: `[]` | `[0.1 1]` | `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, `'[0.1 1]'` for Check Bode Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Magnitudes (dB)

Magnitude values for one or more lower magnitude bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)**.

Settings

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

Tips

- To assert that magnitude bounds are satisfied, select both **Include lower magnitude bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `LowerBoundMagnitudes`
Type: string
Value: `[] | [20 20] | finite numbers | 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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

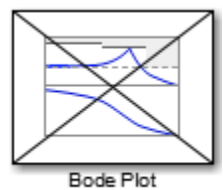
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `'' | MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

  You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Bode Characteristics

Tutorials

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-47
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-69
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

Check Bode Characteristics

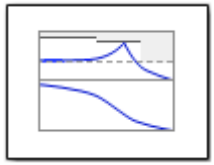
Purpose

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 8-3 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**

Task	Parameters
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**
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

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25

- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-82 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**
	Specify settings.	In **Linearizations** tab: • **Linearize on**

Task		Parameters
		• **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**

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

Gain and Phase Margin Plot

Tutorials

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25
- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-132 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**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times**

Task		Parameters
		• **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.

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

Linear Step Response Plot

Tutorials

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25
- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-182 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**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times**

Task		Parameters
		• **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**

Task	Parameters
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

Nichols Plot

Tutorials

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25
- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-243 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**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**

Task		Parameters
	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**

Task	Parameters
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

Pole-Zero Plot

Tutorials

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25
- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-300 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**

Task		Parameters
		• **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

- "Model Verification at Default Simulation Snapshot Time" on page 5-6
- "Model Verification at Multiple Simulation Snapshots" on page 5-15
- "Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks" on page 5-25
- Verifying Frequency-Domain Characteristics of an Aircraft

How To

"Monitoring Linear System Characteristics in Simulink Models" on page 5-2

Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab:

Task		Parameters
		• "Enable zero-crossing detection" on page 8-315. • "Use exact delays" on page 8-317. • "Linear system sample time" on page 8-318. • "Sample time rate conversion method" on page 8-320. • "Prewarp frequency (rad/s)" on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 8-324. • "Use bus signal names" on page 8-326.
Specify plot type for viewing gain and phase margins.		"Plot type" on page 8-127.
Plot the linear system.		**Show Plot**

Task	Parameters
Specify the feedback sign for closed-loop gain and phase margins.	"Feedback sign" on page 8-114 in **Bounds** tab.
(Optional) Specify bounds on gain and phase margins of the linear system for assertion.	"Include gain and phase margins in assertion" on page 8-110 in **Bounds** tab.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 8-351.

Linearization inputs/outputs

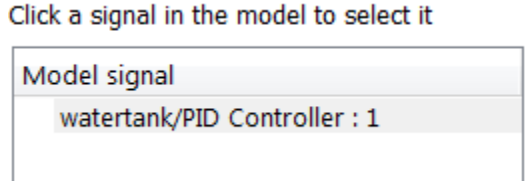
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** area and a new button.

2. Select a signal in the Simulink Editor.

   The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select an individual element.

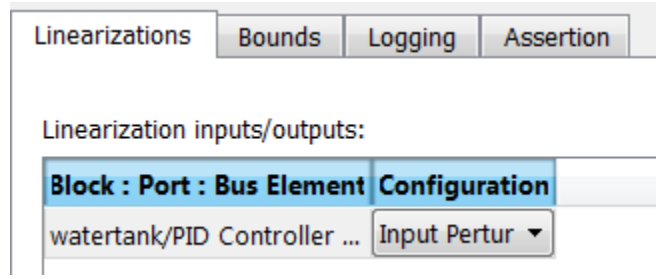
Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression.

To modify the filtering options, click .

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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”.

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

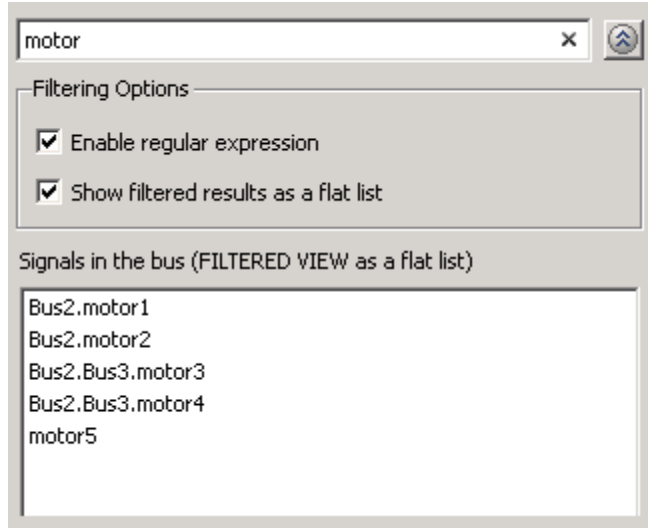
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

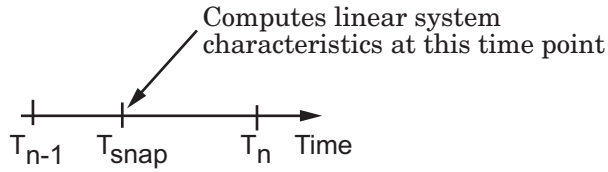
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

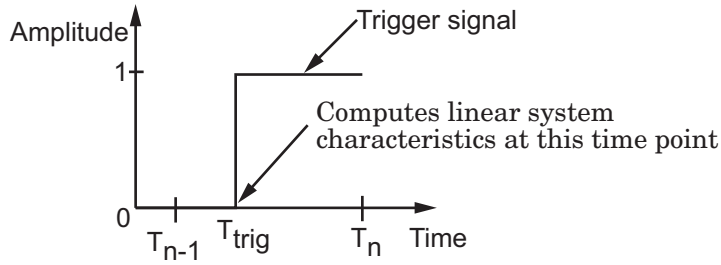
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`

Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include gain and phase margins in assertion

Check that the gain and phase margins are greater than the values specified in **Gain margin (dB) >** and **Phase margin (deg) >**, 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**, is used to compute the margins.

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

You can view the gain and phase margin bound on one of the following plot types:

- Bode
- Nichols
- Nyquist
- Table

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

Settings

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.

Tips

- 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**.

Command-Line Information

Parameter: `EnableMargins`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Gain and Phase Margin Plot block, `'on'` for Check Gain and Phase Margins block

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Gain margin (dB) >

Gain margin, specified in decibels.

By default, negative feedback, specified in **Feedback sign**, is used to compute the gain margin.

You can specify only one gain margin bound on the linear system in this block.

Settings

Default:

`[]` for Gain and Phase Margin Plot block.
`20` for Check Gain and Phase Margins block.

Positive finite number.

Tips

- To assert that the gain margin is satisfied, select both **Include gain and phase margins in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `GainMargin`
Type: string
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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Phase margin (deg) >

Phase margin, specified in degrees.

By default, negative feedback, specified in **Feedback sign**, is used to compute the phase margin.

You can specify only one phase margin bound on the linear system in this block.

Settings

Default:

`[]` for Gain and Phase Margin Plot block.
`30` for Check Gain and Phase Margins block.

Positive finite number.

Tips

- To assert that the phase margin is satisfied, select both **Include gain and phase margins in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `PhaseMargin`
Type: string
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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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 `+`.

Command-Line Information

Parameter: `FeedbackSign`
Type: string
Value: `'-1' | '+1'`
Default: `'-1'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

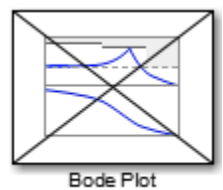
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `'' | MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Plot type

Plot to view gain and phase margins of the linear system computed during simulation.

Settings

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.

Command-Line Information

Parameter: `PlotType`
Type: string
Value: `'bode' | 'nichols' | 'nyquist' | 'table'`
Default: `'bode'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Gain and Phase Margins

Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

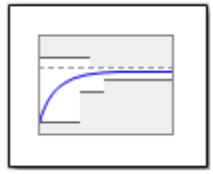
Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing

Task		Parameters
		detection” on page 8-315. • “Use exact delays” on page 8-317. • “Linear system sample time” on page 8-318. • “Sample time rate conversion method” on page 8-320. • “Prewarp frequency (rad/s)” on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • “Use full block names” on page 8-324. • “Use bus signal names” on page 8-326.
Plot the linear system.		**Show Plot**
(Optional) Specify bounds on step response of the linear system for assertion.		Include step response bound in assertion in **Bounds** tab.

Task	Parameters
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • "Enable assertion" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 8-351.

Linearization inputs/outputs

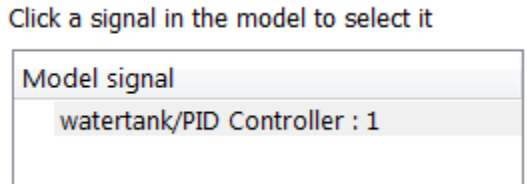
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1 Click .

The dialog box expands to display a **Click a signal in the model to select it** area and a new « button.

2 Select a signal in the Simulink Editor.

The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select an individual element.

Tip For large buses, 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.

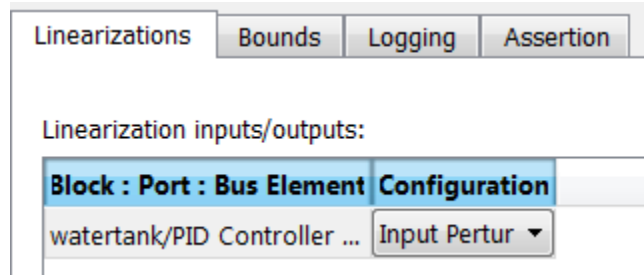
To modify the filtering options, click ».

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

---

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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".

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

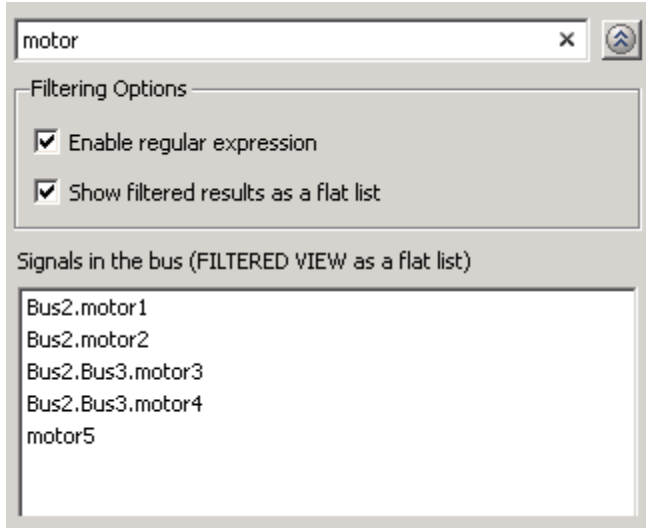
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

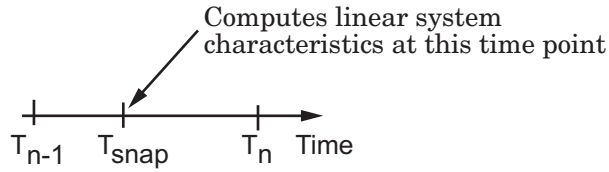
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

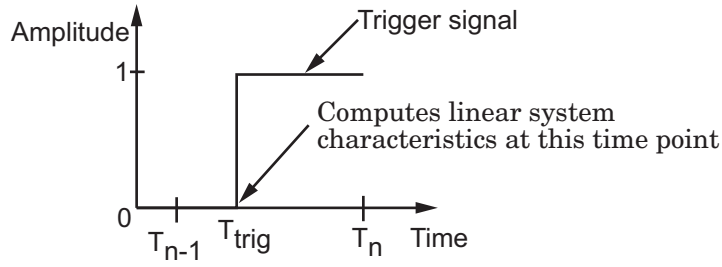
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`
Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Linear Step Response Plot

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`

Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include step response bound in assertion

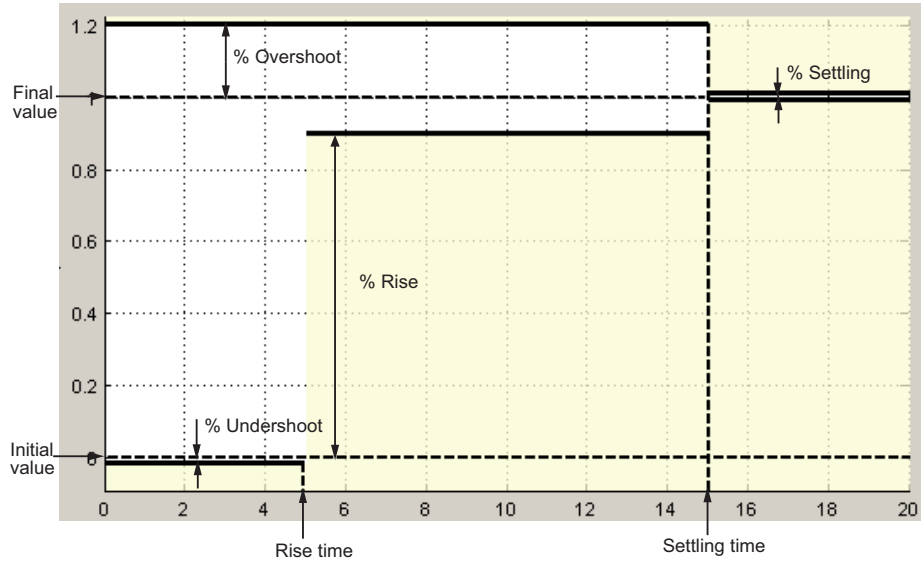
Check that the linear step response satisfies *all* the characteristics specified in:

- **Final value**
- **Rise time** and **% Rise**
- **Settling time** and **% Settling**
- **% Overshoot**
- **% Undershoot**

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

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

The bounds also appear on the step response plot, as shown in the next figure.



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

Settings

Default:

- 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.

Tips

- Clearing this parameter disables the step response bounds and the software stops checking that the bounds are satisfied during simulation. The bound segments are also greyed out on the plot.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableStepResponseBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Linear Step Response Plot block, `'on'` for Check Linear Step Response Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Final value

Final value of the output signal level in response to a step input.

Settings

Default:

- `[]` for Linear Step Response Plot block
- `1` for Check Linear Step Response Characteristics block

Finite real scalar.

Tips

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

Command-Line Information

Parameter: `FinalValue`
Type: string
Value: `[] | 1 | finite real scalar`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Rise time

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

Settings

Default:

- `[]` for Linear Step Response Plot block
- `5` for Check Linear Step Response Characteristics block

Finite positive real scalar, less than the settling time.

Tips

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

Command-Line Information

Parameter: `RiseTime`
Type: string
Value: `[] | 5 | finite positive real scalar`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'5'` for Check Linear Step Response Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

% Rise

The percentage of final value used with the **Rise time**.

Settings

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).

Tips

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

Command-Line Information

Parameter: `PercentRise`
Type: string
Value: `[] | 80 | positive scalar 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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default:

- `[]` for Linear Step Response Plot block
- `7` for Check Linear Step Response Characteristics block

Finite positive real scalar, greater than rise time.

Tips

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

Command-Line Information

Parameter: `SettlingTime`
Type: string
Value: `[] | 7 | positive finite real scalar`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Linear Step Response Plot block, `'7'` for Check Linear Step Response Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

% Settling

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

Settings

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) and less than % overshoot.

Tips

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

Command-Line Information

Parameter: `PercentSettling`
Type: string
Value: `[] | 1 | real 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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

% Overshoot

The amount by which the step response can exceed the final value, specified as a percentage.

Settings

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `10` for Check Linear Step Response Characteristics block

Real number, greater than % settling.

Tips

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

Command-Line Information

Parameter: `PercentOvershoot`
Type: string
Value: `[] | 10 | 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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

% Undershoot

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

Settings

Default:

Minimum: 0

Maximum: 100

- `[]` for Linear Step Response Plot block
- `1` for Check Linear Step Response Characteristics block

Real number.

Tips

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

Command-Line Information

Parameter: `PercentUndershoot`
Type: string
Value: `[] | 1 | real value between 0 and 100`. Must be specified inside single quotes (`''`).
Default: `'1'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`' '`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

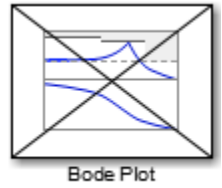
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `''` | `MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Linear Step Response Characteristics

Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

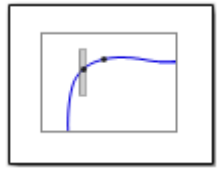
Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing

Task		Parameters
		detection” on page 8-315. • “Use exact delays” on page 8-317. • “Linear system sample time” on page 8-318. • “Sample time rate conversion method” on page 8-320. • “Prewarp frequency (rad/s)” on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • “Use full block names” on page 8-324. • “Use bus signal names” on page 8-326.
Plot the linear system.		**Show Plot**
Specify the feedback sign for closed-loop gain and phase margins.		“Feedback sign” on page 8-226 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. • 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" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.

Task	Parameters
Save linear system to MATLAB workspace.	"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 8-351.

Linearization inputs/outputs

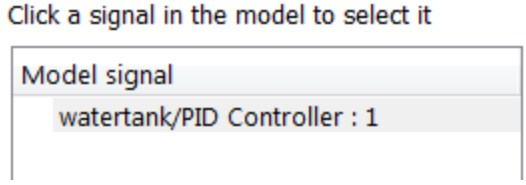
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** area and a new button.

2. Select a signal in the Simulink Editor.

   The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select an individual element.

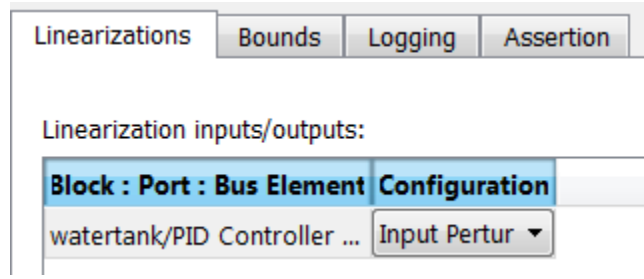
Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression.

To modify the filtering options, click .

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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”.

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

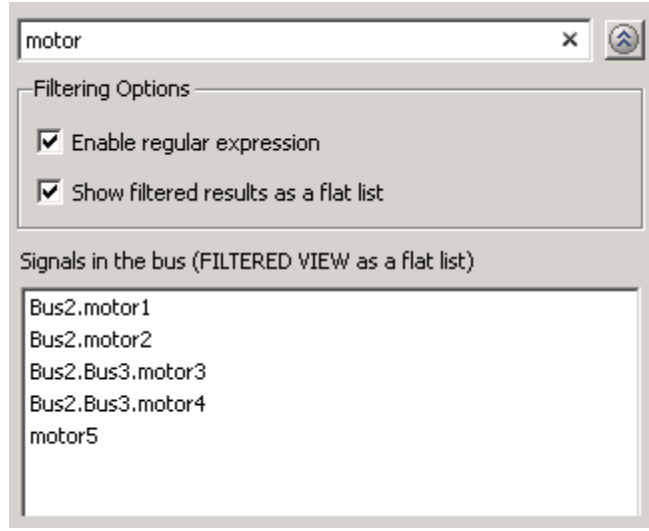
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

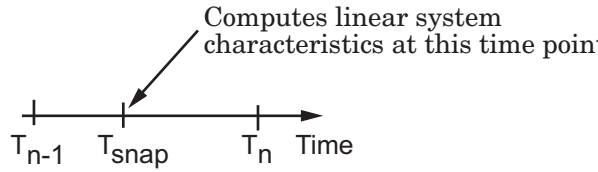
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

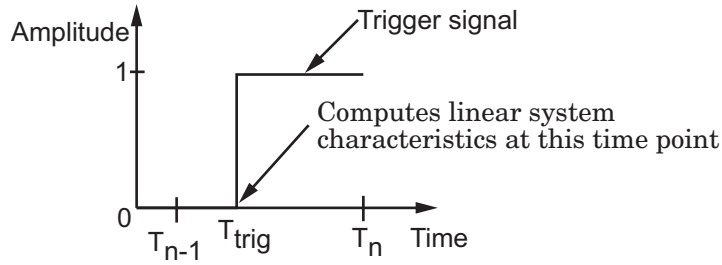
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`

Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include gain and phase margins in assertion

Check that the gain and phase margins are greater than the values specified in **Gain margin (dB) >** and **Phase margin (deg) >**, 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**, is used to compute the margins.

This parameter is used for assertion only if **Enable assertion** 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.

Settings

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.

Tips

- 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**.

Command-Line Information

Parameter: `EnableMargins`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Nichols Plot block, `'on'` for Check Nichols Characteristics block

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Gain margin (dB) >

Gain margin, in decibels.

By default, negative feedback, specified in **Feedback sign**, is used to compute the gain margin.

Settings

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.

Tips

- To assert that the gain margin is satisfied, select both **Include gain and phase margins in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `GainMargin`
Type: string
Value: `[] | 20 | positive finite value`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Nichols Plot block, `'20'` for Check Nichols Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Phase margin (deg) >

Phase margin, in degrees.

By default, negative feedback, specified in **Feedback sign**, is used to compute the phase margin.

Settings

`[]` 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.

Tips

- To assert that the phase margin is satisfied, select both **Include gain and phase margins in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `PhaseMargin`
Type: string
Value: `[] | 30 | positive finite value`. Must be specified inside single quotes (`' '`).
Default: `'[]'` for Nichols Plot block, `'30'` for Check Nichols Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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) <**, 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**, is used to compute the closed-loop peak gain.

This parameter is used for assertion only if **Enable assertion** 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.

Settings

Default: Off

On

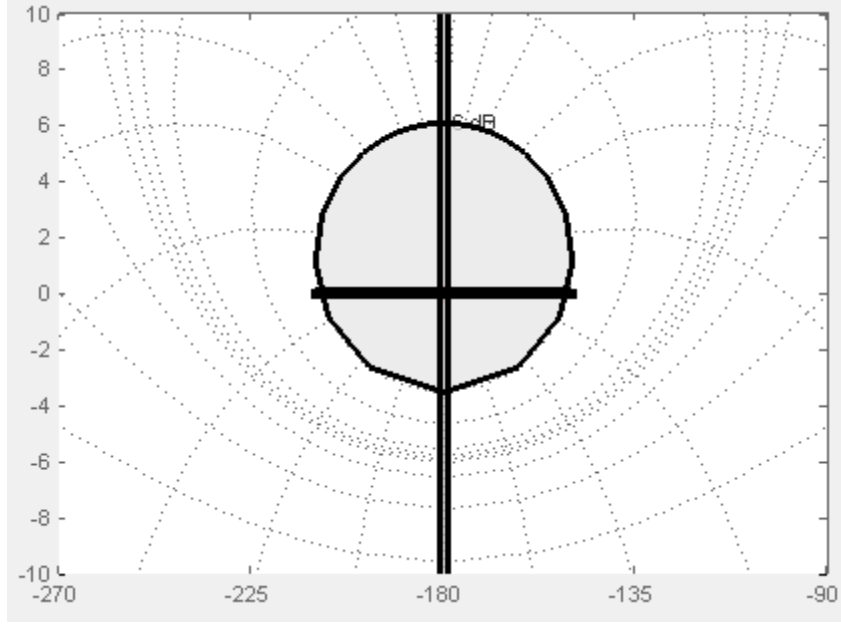
Check that the closed-loop peak gain satisfies the specified value, during simulation.

Off

Do not check that the closed-loop peak gain satisfies the specified value, during simulation.

Tips

- 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**.

Command-Line Information

Parameter: `EnableCLPeakGain`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Closed-loop peak gain (dB) <

Closed-loop peak gain, in decibels.

By default, negative feedback, specified in **Feedback sign**, is used to compute the margins.

Settings

Default `[]`

- Positive or negative finite number for one bound.
- Cell array of positive or negative finite numbers for multiple bounds.

Tips

- To assert that the gain margin is satisfied, select both **Include closed-loop peak gain in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `CLPeakGain`
Type: string
Value: `[] | positive or negative number | cell array of positive or negative numbers`. Must be specified inside single quotes (`' '`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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)** and **Open-loop gains (dB)**, during simulation. The software displays a warning if the Nichols response violates the bounds.

This parameter is used for assertion only if **Enable assertion** 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.

Settings

Default: Off

On

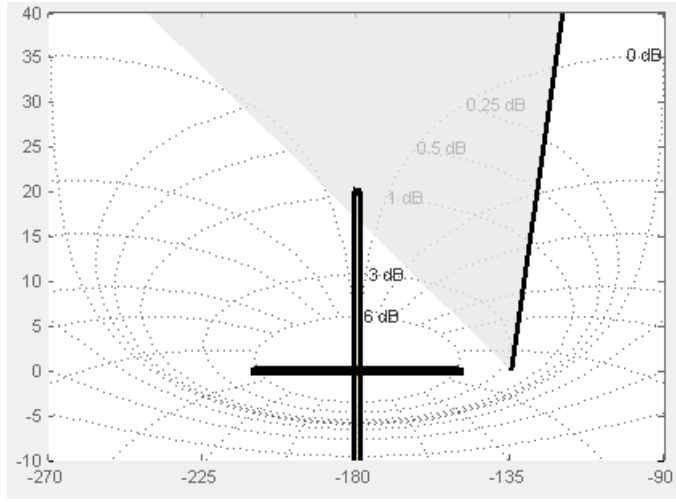
Check if the Nichols response satisfies the specified open-loop gain and phase bounds, during simulation.

Off

Do not check if the Nichols response satisfies the specified open-loop gain and phase bounds, during simulation.

Tips

- 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**.

Command-Line Information

Parameter: `EnableGainPhaseBound`
Type: string
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)**.

Settings

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

Tips

- To assert that the open-loop gains and phases are satisfied, select both **Include open-loop gain-phase bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `OLPhases`
Type: string
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: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Open-loop gains (dB)

Open-loop gains, in decibels.

Specify the corresponding open-loop phases in **Open-loop phases (deg)**.

Settings

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

Tips

- To assert that the open-loop gains are satisfied, select both **Include open-loop gain-phase bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `OLGains`
Type: string

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: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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 `+`.

Command-Line Information

Parameter: `FeedbackSign`
Type: string
Value: `'-1' | '+1'`
Default: `'-1'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

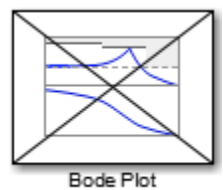
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `'' | MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Nichols Characteristics

Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

Pole-Zero Plot

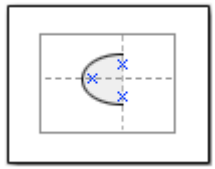
Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing

Task		Parameters
		detection” on page 8-315. • “Use exact delays” on page 8-317. • “Linear system sample time” on page 8-318. • “Sample time rate conversion method” on page 8-320. • “Prewarp frequency (rad/s)” on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • “Use full block names” on page 8-324. • “Use bus signal names” on page 8-326.
Plot the linear system.		**Show Plot**

Task	Parameters
(Optional) Specify bounds on pole-zero 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" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.

Task	Parameters
Save linear system to MATLAB workspace.	"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 8-351.

Linearization inputs/outputs

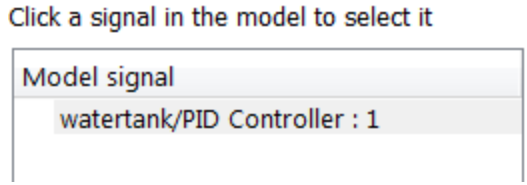
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** area and a new button.

2. Select a signal in the Simulink Editor.

   The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select an individual element.

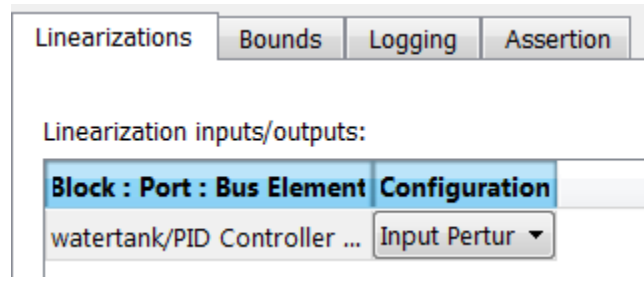
Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression.

To modify the filtering options, click .

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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".

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

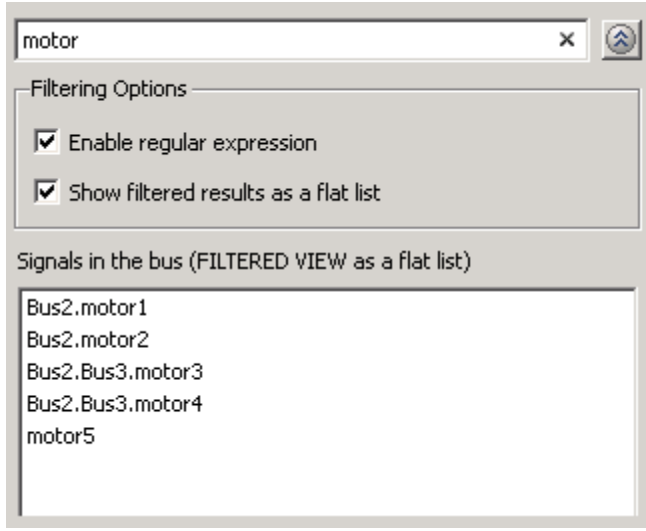
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

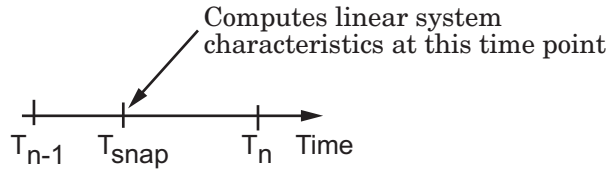
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

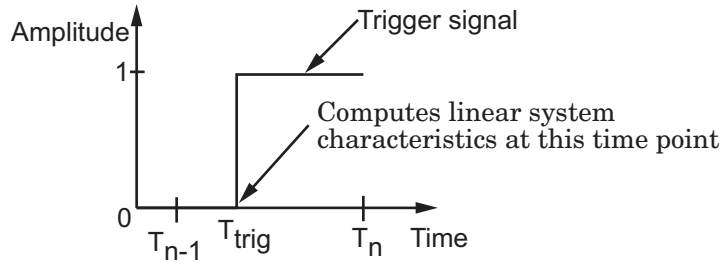
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`

Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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) <=**. 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** 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.

Settings

Default:

- Off for Pole-Zero Plot block.
- On for Check Pole-Zero Characteristics block.

On

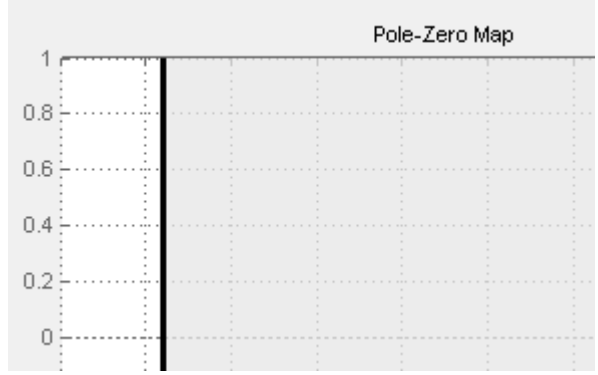
Check that each pole lies in the region defined by the settling time bound, during simulation.

Off

Do not check that each pole lies in the region defined by the settling time bound, during simulation.

Tips

- 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, damping ratio or natural frequency, but want to exclude the settling time bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableSettlingTime`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Pole-Zero Plot block, `'on'` for Check Pole-Zero Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Settling time (sec) <=

Settling time, in seconds, of the second-order system.

Settings

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.

Tips

- To assert that the settling time bounds are satisfied, select both **Include settling time bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `SettlingTime`
Type: string
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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include percent overshoot bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the percent overshoot, specified in **Percent overshoot <=**. 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** 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.

Settings

Default:

Off for Pole-Zero Plot block.
On for Check Pole-Zero Characteristics block.

On

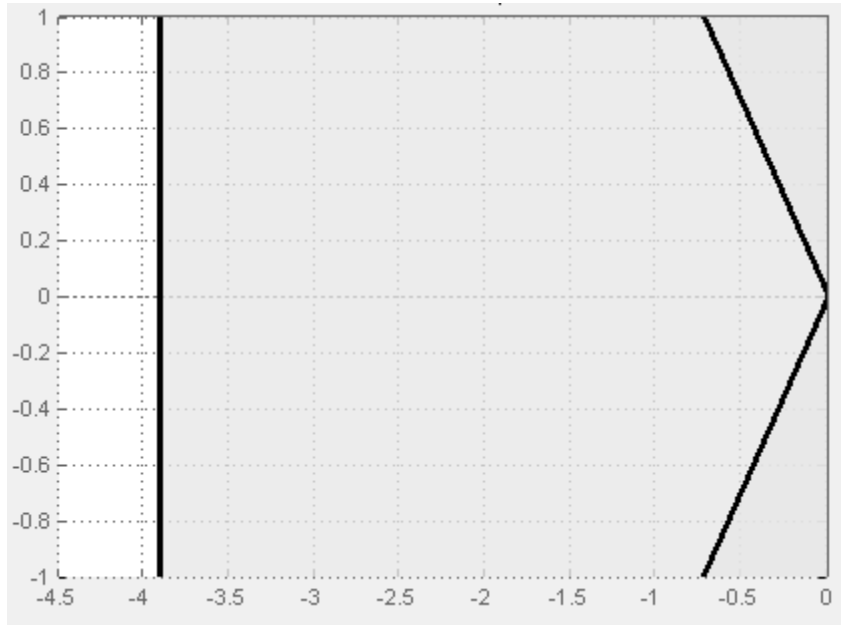
Check that each pole lies in the region defined by the percent overshoot bound, during simulation.

Off

Do not check that each pole lies in the region defined by the percent overshoot bound, during simulation.

Tips

- 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, damping ratio or natural frequency, but want to exclude the percent overshoot bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnablePercentOvershoot`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Pole-Zero Plot block, `'on'` for Check Pole-Zero Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Percent overshoot <=

Percent overshoot of the second-order system.

Settings
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.

Tips

- The percent overshoot *p.o* can be expressed in terms of the damping ratio 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** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `PercentOvershoot`

Type: string

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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include damping ratio bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the damping ratio, specified in **Damping ratio >=**. 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** 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.

Settings

Default: Off

On

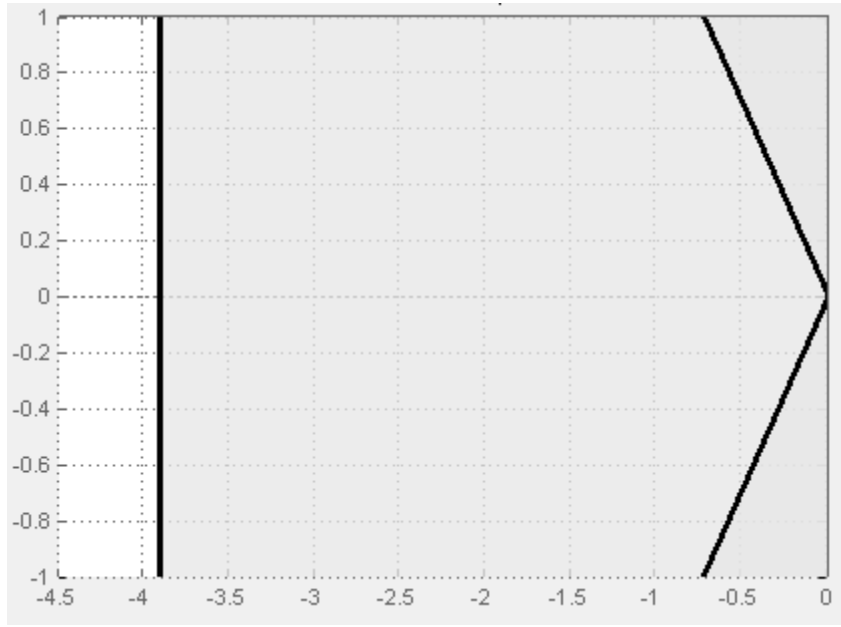
Check that each pole lies in the region defined by the damping ratio bound, during simulation.

Off

Do not check that each pole lies in the region defined by the damping ratio bound, during simulation.

Tips

- 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, percent overshoot or natural frequency, but want to exclude the damping ratio bound from assertion, clear this parameter.
- To only view the bounds on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableDampingRatio`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Damping ratio >=

Damping ratio of the second-order system.

Settings

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.

Tips

- 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** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `DampingRatio`
Type: string

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: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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)**. 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** 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.

Settings

Default: Off

On

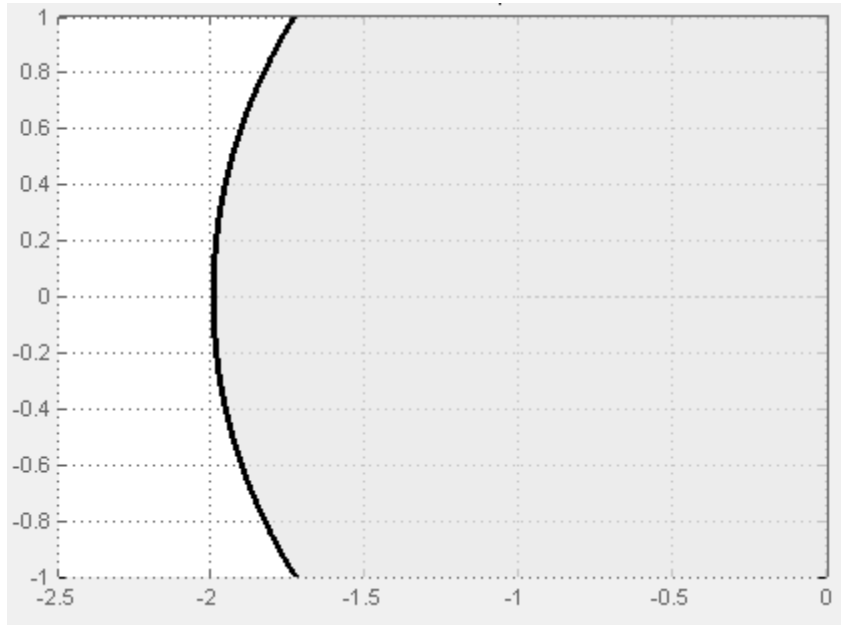
Check that each pole lies in the region defined by the natural frequency bound, during simulation.

Off

Do not check that each pole lies in the region defined by the natural frequency bound, during simulation.

Tips

- 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, percent overshoot or damping ratio 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**.

Command-Line Information

Parameter: `NaturalFrequencyBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Natural frequency (rad/sec)

Natural frequency of the second-order system.

Settings

Default: `[]`

- Finite positive real scalar for single natural frequency bound.
- Cell array of finite positive real scalars for multiple natural frequency bounds.

Tips

- To assert that the natural frequency bounds are satisfied, select both **Include natural frequency bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `NaturalFrequency`
Type: string
Value: `[] | positive finite real scalar | cell array of positive finite real scalars`. Must be specified inside single quotes (`' '`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

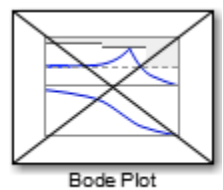
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `'' | MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

  You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Pole-Zero Characteristics

Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47
- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

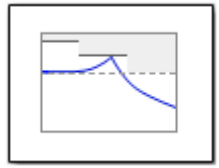
Purpose

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 8-303. • "Click a signal in the model to select it" on page 8-307.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 8-311. • "Snapshot times" on page 8-313. • "Trigger type" on page 8-314.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab:

Task		Parameters
		• "Enable zero-crossing detection" on page 8-315. • "Use exact delays" on page 8-317. • "Linear system sample time" on page 8-318. • "Sample time rate conversion method" on page 8-320. • "Prewarp frequency (rad/s)" on page 8-323.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 8-324. • "Use bus signal names" on page 8-326.
Plot the linear system.		**Show Plot**

Task	Parameters
(Optional) Specify bounds on singular values 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" on page 8-345. • "Simulation callback when assertion fails (optional)" on page 8-347. • "Stop simulation when assertion fails" on page 8-348. • "Output assertion signal" on page 8-349.
Save linear system to MATLAB workspace.	"Save data to workspace" on page 8-340 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	"Show plot on block open" on page 8-351.

Linearization inputs/outputs

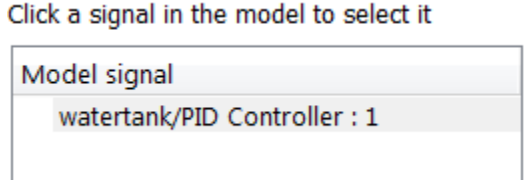
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** area and a new button.

2. Select a signal in the Simulink Editor.

   The selected signal appears as a **Model signal** in the **Click a signal in the model to select it** area.

3. (Optional) For buses, expand the bus signal to select an individual element.

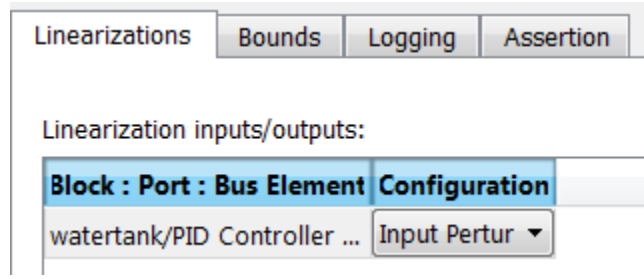
Tip For large buses, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter MATLAB regular expression.

To modify the filtering options, click .

Filtering Options

- "Enable regular expression" on page 8-308
- "Show filtered results as a flat list" on page 8-309

4 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 .

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.

Settings

No default

Command-Line Information

Use `getlinio` and `setlinio` to specify linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

- changes to .

  Use to collapse the **Click a signal in the model to select it** area.

Settings

No default

Command-Line Information

Use the `getlinio` and `setlinio` commands to select signals as linearization inputs and outputs.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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”.

Settings

Default: On

On

Allow use of MATLAB regular expressions for filtering signal names.

Off

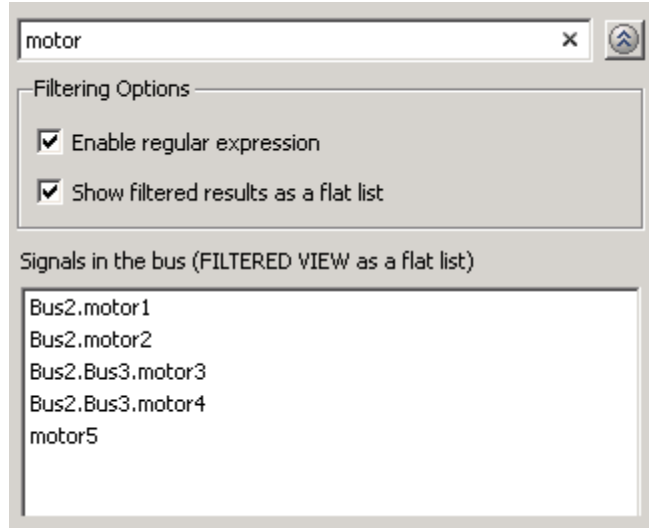
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal string.

Dependencies

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.



Settings

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.

Dependencies

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.

Settings

Default: `Simulation snapshots`

`Simulation snapshots`
: Specific simulation time, specified in **Snapshot times**.

  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**.

  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.

Dependencies

- Setting this parameter to `Simulation snapshots` enables **Snapshot times**.
- Setting this parameter to `External trigger` enables **Trigger type**.

Command-Line Information

Parameter: `LinearizeAt`

Type: string
Value: `'SnapshotTimes' | 'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Snapshot times

One or more simulation times. The linear system is computed at these times.

Settings

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.

Dependencies

Selecting `Simulation snapshots` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: string
Value: `0 | positive real number | vector of positive real numbers`
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Trigger type

Trigger type of an external trigger for computing linear system.

Settings

Default: `Rising edge`

`Rising edge`
: Rising edge of the external trigger signal.

`Falling edge`
: Falling edge of the external trigger signal.

Dependencies

Selecting `External trigger` in **Linearize on** enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: string
Value: `'rising' | 'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

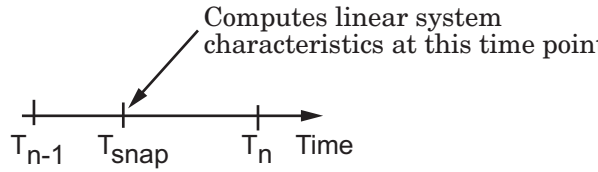
"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

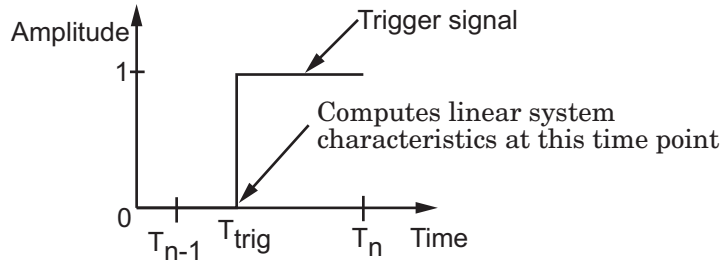
- The exact snapshot times, specified in **Snapshot times**.

  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**.

  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" in the *Simulink User Guide*.

Settings

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.

Command-Line Information

Parameter: `ZeroCross`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseExactDelayModel`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

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.

Command-Line Information

Parameter: `SampleTime`

Type: string
Value: `auto` | `Positive finite value` | `0`
Default: `auto`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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** is not `auto`.

Settings

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" 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" 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" 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.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)**.

Command-Line Information

Parameter: `RateConversionMethod`
Type: string
Value: `'zoh' | 'tustin' | 'prewarp'| 'upsampling_zoh'| 'upsampling_tustin'| 'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Prewarp frequency (rad/s)

Prewarp frequency for Tustin method, specified in radians/second.

Settings

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.

Dependencies

Selecting either

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

in **Sample time rate conversion method** enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: string
Value: `10 | positive scalar value`
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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`

Settings

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`.

Command-Line Information

Parameter: `UseFullBlockNameLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Command-Line Information

Parameter: `UseBusSignalLabels`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include upper singular value bound in assertion

Check that the singular values satisfy upper bounds, specified in **Frequencies (rad/sec)** and **Magnitude (dB)**, 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** 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.

Settings

Default:

- Off for Singular Value Plot block.
- On for Check Singular Value Characteristics block.

On

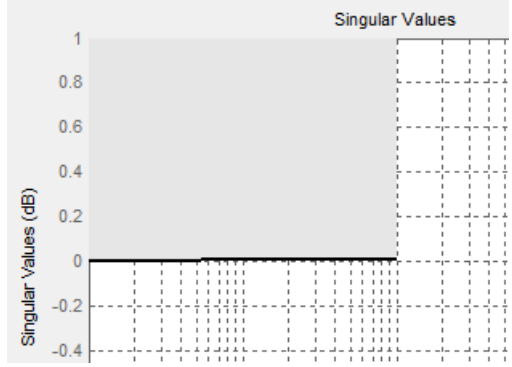
Check that the singular value satisfies the specified upper bounds, during simulation.

Off

Do not check that the singular value satisfies the specified upper bounds, during simulation.

Tips

- 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 but want to include only the lower bounds for assertion, clear this parameter.
- To only view the bound on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableUpperBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'` for Singular Value Plot block, `'on'` for Check Singular Value Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Frequencies (rad/sec)

Frequencies for one or more upper singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)**.

Settings

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.

Tips

- To assert that magnitudes that correspond to the frequencies are satisfied, select both **Include upper singular value bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `UpperBoundFrequencies`
Type: string
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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Magnitudes (dB)

Magnitude values for one or more upper singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)**.

Settings

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

Tips

- To assert that magnitudes are satisfied, select both **Include upper singular value bound in assertion** and **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 `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.

Command-Line Information

Parameter: `UpperBoundMagnitudes`

Type: string

Value: `[]` | `[0 0]` | `finite numbers` | `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.

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Include lower singular value bound in assertion

Check that the singular values satisfy lower bounds, specified in **Frequencies (rad/sec)** and **Magnitude (dB)**, 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** 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.

Settings

Default: Off

On

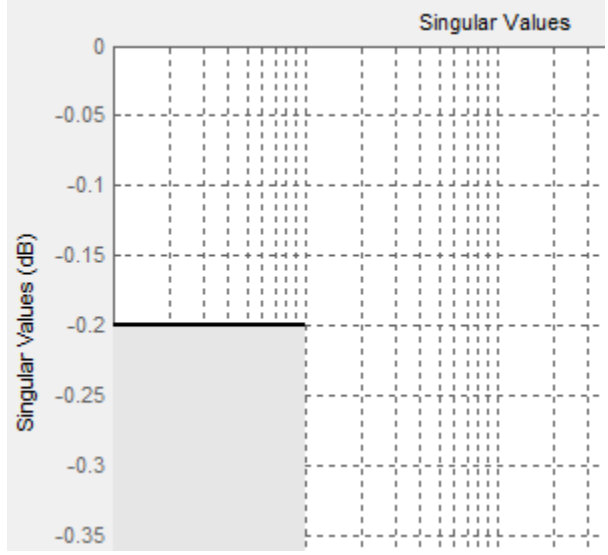
Check that the singular value satisfies the specified lower bounds, during simulation.

Off

Do not check that the singular value satisfies the specified lower bounds, during simulation.

Tips

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



- If you specify both lower and upper singular value bounds but want to include only the upper bounds for assertion, clear this parameter.
- To only view the bound on the plot, clear **Enable assertion**.

Command-Line Information

Parameter: `EnableLowerBound`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Frequencies (rad/sec)

Frequencies for one or more lower singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)**.

Settings

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.

Tips

- To assert that magnitude bounds that correspond to the frequencies are satisfied, select both **Include lower singular value bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

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

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Magnitudes (dB)

Magnitude values for one or more lower singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)**.

Settings

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

Tips

- To assert that magnitudes are satisfied, select both **Include lower singular value bound in assertion** and **Enable assertion**.
- 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.

Command-Line Information

Parameter: `LowerBoundFrequencies`
Type: string

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

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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 objects.
- `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 **Save Simulation output as single object**.

For more information about data logging in Simulink, see "Export Simulation Data" and the `Simulink.SimulationOutput` reference page.

Settings

Default: Off

On

Save the computed linear system.

Off

Do not save the computed linear system.

Dependencies

This parameter enables **Variable name**.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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" and the `Simulink.SimulationOutput` reference page.

Settings

Default: sys

String.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: string
Value: `sys | any string`. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

Default: Off

On

Save the operating points.

Off

Do not save the operating points.

Dependencies

Save data to workspace enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Enable assertion

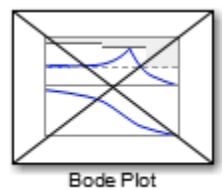
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

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

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

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

Settings

Default: On

On

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

Off

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

Dependencies

This parameter enables:

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

Command-Line Information

Parameter: `enabled`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

No Default

A MATLAB expression.

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `callback`
Type: string
Value: `'' | MATLAB expression`
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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.

Settings

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.

Tips

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

Dependencies

Enable assertion enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

“Model Verification at Default Simulation Snapshot Time” on page 5-6

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

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

Settings

Default: Off

On

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

Off

Do not output a Boolean signal to indicate assertion status.

Tips

- Use this parameter to design complex assertion logic. For an example, see “Model Verification Using Simulink® Control Design™ and Simulink Verification Blocks” on page 5-25.

Command-Line Information

Parameter: `export`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

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**.

Settings

Default: Off

On

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

Off

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

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation

"Model Verification at Default Simulation Snapshot Time" on page 5-6

Show Plot

Open the plot window.

Use the plot to view:

- Linear system characteristics computed from the nonlinear Simulink model during simulation

You must click this button before you simulate the model to view the linear characteristics.

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

- Bounds on the linear system characteristics

  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 on each plot, see "Verifiable Linear System Characteristics" on page 5-5 in the User's Guide.

  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 Simulink Editor active and highlights the block.
- Simulating the model by clicking or selecting **Simulation > Run**. 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 .

See Also

Check Singular Value Characteristics

Tutorials

- "Visualize Bode Response of Simulink Model During Simulation" on page 2-47

- "Visualize Linear System at Multiple Simulation Snapshots" on page 2-69
- "Visualize Linear System of a Continuous-Time Model Discretized During Simulation" on page 2-75
- Plotting Linear System Characteristics of a Chemical Reactor

Trigger-Based Operating Point Snapshot

Purpose

Generate operating points, linearizations, or both at triggered events

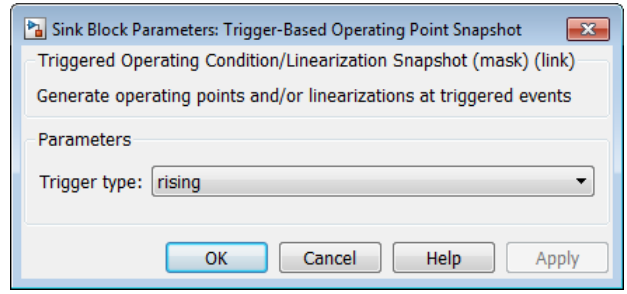
Library

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`

9

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 3-28
- "Effects of Time-Varying Source Blocks on Frequency Response Estimation" on page 3-63

- `frest.findSources` reference page
- `frestimateOptions` reference page
- `frestimate` reference page

Index

A

add linearization input and output 8-5 8-85 8-134 8-185 8-246 8-303
`addoutputspec` function 7-2

B

bus signal names 8-28 8-108 8-155 8-208 8-269 8-326

C

closed-loop gain on Nichols plot 8-218
compensator design
  analysis plots 4-82
  beginning a task 4-68
  closed-loop responses 4-73
  creating a SISO Design Task 4-78
  design methods 4-89
  design plots 4-79
  open-loop systems 4-79
  operating points 4-75
  overview 4-2
  retrieving designs 4-94
  storing designs 4-94
  time delays 4-88
  tunable blocks 4-70
  writing to Simulink model 4-96
configuration parameters
  pane
    Open-loop gains (dB): 8-224
Control and Estimation Tools Manager
  compensator design task 4-68
  SISO Design Task 4-78
`copy` function 7-6

D

damping ratio on pole-zero plot 8-280
delays
  compensator design 4-88

E

enable assertion 8-47 8-121 8-172 8-233 8-291 8-345
enable zero-crossing detection 8-17 8-97 8-146 8-197 8-258 8-315
equilibrium operating point 1-2
  from simulation 1-40
  from specifications 1-14
equilibrium states
  from simulation 1-40
  from specifications 1-14

F

final value of step response 8-160
frequencies for lower magnitude bound 8-38
frequencies for lower singular value bound 8-336
frequencies for upper magnitude bound 8-32
frequencies for upper singular value bound 8-330

G

gain margin bound 8-112
gain margin bound on Nichols plot 8-212
`get` function 7-79
`getinputstruct` function 7-82
`getlinio` function 7-83
`getlinplant` function 7-92
`getstatestruct` function 7-96
`getxu` function 7-98

I

include closed-loop peak gain 8-216
include damping ratio on pole-zero plot 8-278
include gain and phase margins 8-110
include gain and phase margins on Nichols plot 8-210

include lower magnitude bound 8-36
include lower singular value bound 8-334
include natural frequency on pole-zero plot 8-282
include open-loop gain-phase bound 8-220
include percent overshoot bound on pole-zero plot 8-274
include settling time bound on pole-zero plot 8-271
include step response bound 8-157
include upper magnitude bound 8-30
include upper singular value bound 8-328
`initopspec` function 7-101

L

linearization
using simulation event 2-65
using simulation snapshot 2-62
linearize on 8-13 8-93 8-142 8-193 8-254 8-311
`linio` function 7-127
`linoptions` function 7-138
lower magnitude bound 8-40
lower singular value bound magnitude 8-338

O

open-loop phases 8-222
operating point
at simulation snapshot 1-40
equilibrium 1-2
from simulation snapshot 1-40
initialize simulation 1-43
initializing optimization search 1-24
meet output specifications 1-21
optimization settings 1-34
SimMechanics model 1-29
steady state 1-2
steady states from simulation 1-40
steady states from specifications 1-14
trimming 1-14
`operpoint` function 7-145
output assertion signal 8-51 8-125 8-176 8-237 8-295 8-349

P

percent overshoot 8-165
percent overshoot on pole-zero plot 8-276
percent rise 8-162
percent settling 8-164
percent undershoot 8-166
phase margin bound 8-113 to 8-114
phase margin bound on Nichols plot 8-214
PID
automatic tuning 4-3
plot type for gain and phase margins 8-127
prewarp frequency 8-25 8-105 8-153 8-205 8-266 8-323

R

retrieving compensator designs 4-94
rise time 8-161

S

sample time 8-20 8-100 8-148 8-200 8-261 8-318
Sample time rate conversion method 8-22 8-102 8-151 8-202 8-263 8-320
save data to workspace 8-42 8-116 8-167 8-228 8-286 8-340
select signal 8-9 8-89 8-138 8-189 8-250 8-307
`set` function 7-158
`setlinio` function 7-161
settling time 8-163
settling time on pole-zero plot 8-273
`setxu` function 7-166
show plot on block open 8-53 8-128 8-178 8-239 8-296 8-351
simulation
initialize operating point 1-43

simulation callback 8-49 8-123 8-174 8-235 8-293 8-347
SISO Design Task
creating 4-78
Snapshot times 8-15 8-95 8-144 8-195 8-256 8-313
steady state operating point 1-2
from simulation 1-40
from specifications 1-14
Stop simulation when assertion fails 8-50 8-124 8-175 8-236 8-294 8-348
storing compensator designs 4-94

T

trigger type 8-16 8-96 8-145 8-196 8-257 8-314
trim condition 1-2
trimming 1-14

U

`update` function 7-347
upper magnitude bound 8-34
upper singular value bound magnitude 8-332
use exact delays 8-19 8-99 8-148 8-199 8-260 8-317

V

variable name for saved linear system 8-44 8-118 8-169 8-230 8-288 8-342